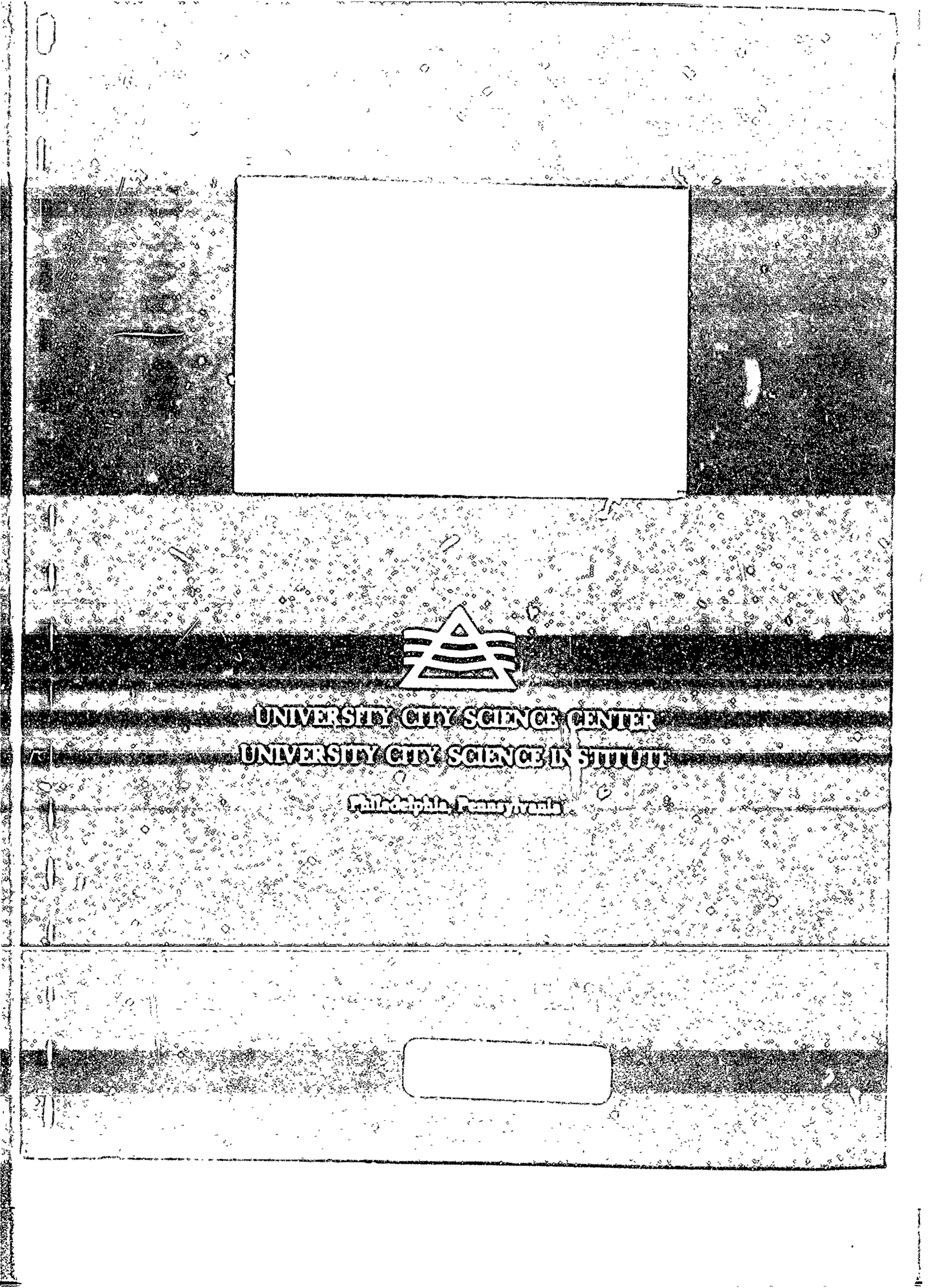
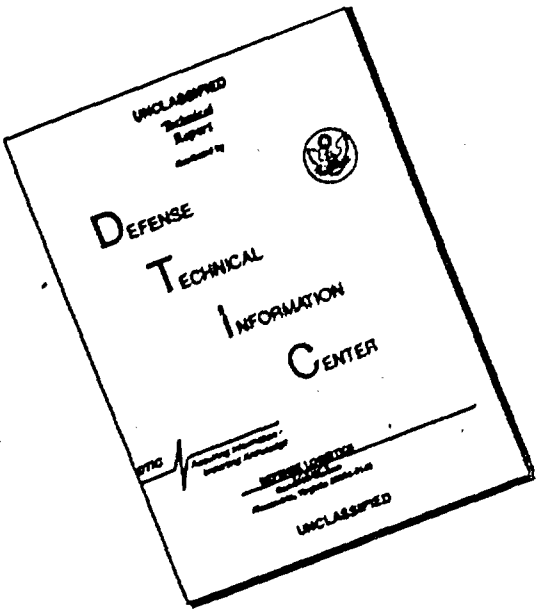

UNIVERSITY CITY SCIENCE CENTER
UNIVERSITY CITY SCIENCE INSTITUTE
Philadelphia, Pennsylvania

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

EXTENSION OF PROGRAMMING LANGUAGE CONCEPTS

Philip R. Bagley

The research work reported herein
was sponsored by the Air Force Office
of Scientific Research, Office of
Aerospace Research, under contract
F44620-67-C-0021.

University City Science Center
3401 Market Street
Philadelphia, Pennsylvania 19104

ABSTRACT

This study concerns the extension of concepts used in current computer programming languages. The aim is to find ways of designing new programming languages having increased flexibility without also having increased complexity. To increase flexibility means to place on the user as few restrictions as possible on what he can express and modify and on the notational conventions he may choose. The key to accomplishing this is by generalizing on the current concepts.

The work is based on the idea that it is possible to design a language which is truly independent of the hardware characteristics of current computers. In the course of the study, considerable re-examination of current concepts such as variables, procedure call mechanisms, and program sequence controls, has been required.

A new technique of expressing data values, data elements, and data structures, has been developed. This technique provides for the expression of: (1) domains of values (familiar examples of which are "real", "integer", and "Boolean"), (2) simple data elements ("variables") which take on a value from some domain of values, (3) composite data elements, which are "associations" of other data elements, and (4) relationships between these data elements, where each such relationship is itself a composite data element. The technique provides for the construction of arbitrarily-complex data elements, and for arbitrarily-chosen relationships between data elements.

All expressions in a program which cause the language processor to take some action, which includes "declarations", are viewed as transformations ("procedures"). A basic set of these transformations has been proposed. The two main classes of transformations are: (1) transformations of data, which create, test, modify, and destroy data elements, and (2) transformations of sequence control, which includes control of iteration, and of conditional execution. Functions (such as "add") are a sub-class of transformations, the sub-class which generate a single result. Functions are definable in two ways: (1) in terms of other transformations, and (2) by enumeration, in the form of function tables. The following components of computer languages are all regarded as forms of data elements: (1) calls to transformations, (2) program sequence controls,

(3) domains of values which are defined by enumeration, and (4) identifiers. Thus each of these components can be manipulated by transformations defined for data.

The most significant demand on machine design which arises from this research is that much more freedom of storage organization is needed than is provided by conventional machines. Large-scale associative memories could be used to provide some of this needed flexibility of storage.

Recommendations for further work are presented and an extensive bibliography on programming language, concepts, and design is appended.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

The goal of this research is to reduce significantly the programming effort and elapsed time required to write and debug computer programs. The research task can be characterized as showing how:

1. To provide the programmer with a wide variety of programming concepts (that is, data structures and operation), at least as wide as the set of concepts collectively available in current programming languages. However, this must be done without creating a language so complex as to be unmanageable. The key is to generalize on the concepts available in current languages, to find more general notions of which the set in current languages is a set of more specific cases.
2. To reduce the amount of detail that a programmer must concern himself with, without preventing him from being able to specify detail when he wishes. One such avenue of simplification is to remove from the programmer's immediate concern all matters of how a program will be implemented on a specific machine.

The elimination of any consideration of a compiling phase contributes to both goals above. First, by so doing, the language is not restricted to expressions which can be compiled; second, the distinction between processes carried out at compile time and those carried out at execute time vanishes, thereby simplifying the intellectual task of learning and dealing with the language.

We can view the research task as one of developing a language for programming a hypothetical machine. We can endow this machine with all the nice characteristics of real machines. We must select for this machine a set of primitive concepts which are in some sense optimally convenient. ("Primitive" means defined in some other way than in terms of the structure and commands for our hypothetical machine.) We note that it is never a matter of being unable to represent some concept or terms of the available primitives: it is only a matter of how much struggle it is.

Devising such a hypothetical machine is not just an academic exercise. We know that with a simulator, or interpretive program, we can make any real machine behave like our hypothetical machine. We must qualify this by adding "with respect to the computational results." The simulated

hypothetical machine might not be "efficient." That is, it might not compute results for a given program as fast as the real machine could be programmed (in assembly language) to do it. This possible loss of efficiency is the price that must be paid for having the luxury of a hypothetical machine which makes it significantly easier for the programmer to do his job. We hasten to add that as our ability to write simulators improves, the inefficiency contributed by the simulator will decrease. We hope also that knowledge of what is needed for a simulator will stimulate developments in machine design which will aid in producing simulators which operate with more "efficiency."

To repeat, the crux of programming language design is to choose a good set of primitives, and to relieve the user of as much detail as he wishes to be relieved of. I do not feel that current languages come close enough to this goal. I hope to be able to show how we can come closer to such a goal.

Some languages, both existing ones and proposed ones, have rather elegant facilities built-in for defining new concepts. But the definitional mechanism is not the issue here. It is rather, how to select a set of building blocks (primitives) out of which "new concepts" can be constructed without agony, circumlocutions, and slightly unsatisfactory substitutes for the data elements and structures that are really wanted.

Although we will be developing many definitions later in this report, it is helpful to give at this point a few informal definitions of terms which will be used in our preliminary discussions:

1. "Language" means a computer programming language representative of the class whose members are the following current advanced programming languages: ALGOL, COBOL, LISP, COMIT, IPL-V, FORTRAN, FACT, SNOBOL, SLIP, JOVIAL, PL/I, and FORMAC.
2. "Problem-solution concept" means the programmer's mental concept of what he wants a program to accomplish. It means the result of any systems analysis or problem-solving activity which he may have done in order to decide in principle what he is to write a program to do. It means a general algorithm, without the complete detail necessary to make it unambiguous and precise enough for computer implementation.

3. "Programming" and "preparing a program" means the total programmer activity, beginning with a "problem solution concept" devising an algorithm, writing it in some language, and debugging it. It also includes these activities applied to the modification of a program already written and debugged. By way of contrast, "programming" as used herein specifically does not include the task usually called "system analysis"—the refining of a statement of a problem or goal until it is clear what is desired, or at least until it is clear what a program is to accomplish.
4. "Programmer" means a person who accomplished programming, but it refers specifically to one who is trained for the programming job, and has several years experience, rather than one who is a casual computer user.
5. "Data elements" means instances of data types, which are such things as numbers, truth values, symbols composed of concatenated characters, strings, and names of other entities.
6. "Data structures" are the conceptual arrangements of data elements, such as in tables, arrays, pushdown lists, and hierarchies. When we need a collective name for "data elements and structures," we say "data objects."

CHAPTER 2. GOALS AND ASSUMPTIONS

STATEMENT OF PROBLEM

Need to reduce programming effort and elapsed time. In the area of writing computer programs there is an undisputed and increasing need to reduce both the programmer effort and the elapsed time required to prepare a program. ("Elapsed time" means the interval between problem definition and the achievement of a correctly-running program.) Furthermore, since the cost of programming is going up with respect to the cost of machine computations and the cost of computation is constantly going down, this need is becoming more intense. In a number of situations today, particularly military command applications, there is a high premium on reducing elapsed time.

Ways to reduce effort and time is not only needed for preparing new programs but also for revising old ones. It is widely, if not universally, recognized that all but the simplest systems must continually evolve if they are to retain their usefulness. A major problem with current computer-based systems is that their capability to evolve—to be modified to keep pace with changing system requirements—is poor. To modify most sizable computer programs, no matter what language they are written in, is a time-consuming, expensive, unpleasant process fraught with errors. From my own experience, I believe it is no exaggeration to say that sizable computer programs, such as those used in military command and control applications, are obsolete before they are finished and that they cannot be modified fast enough to be satisfactory for current needs. I contend that the application of computers to sizable systems is going to be severely handicapped until significant progress is made in reducing programmer and elapsed time.

Major deficiencies of current languages. Many attempts have been made at making programming easier and faster. Among the present languages there have been some modest advances. For many problems, useful and seemingly adequate languages do exist. It is my conviction, however, that a significant advance has been prevented by failure to recognize and overcome several major deficiencies. All current and widely-known languages possess these deficiencies in large degree. These deficiencies will be discussed in turn.

The first major deficiency of current languages is that they are too restricted in what can readily be expressed in them. A programming language reflects a philosophy of the world—it represents the way that the world is viewed, in terms of objects and their possible relationships, and in terms of the possible manipulations upon these. ALGOL and FORTRAN, for example, view the world as describable in terms of 3 sets of atomic objects which can be manipulated as variables:

1. integers less than some magnitude determined by the word length of the machine on which the program is to be run;
2. rational numbers representable with a fixed number of significant digits, determined by the word length and the arithmetic circuitry of the machine on which the program is to be run;
3. Boolean quantities true and false;

and in terms of one relationship: arrangement of homogeneous atomic objects in rectangular arrays. ALGOL and FORTRAN literally do not admit the existence of any other objects for manipulation. (Character strings are permitted only as constants.) Whatever is expressed in computer languages is expressed either in the primitives (basic terms) of the language or in terms defined by these primitives. Every language designer picks what he thinks is a desirable basic set of primitives. From then on users are "stuck" with the choice, unless they wish to "go outside the language." It appears from the present research work that the number of basic data types and structures required for a broadly-applicable language is not large. It is surprising therefore, that the current programming languages do not contain a set which is convenient for wide range of problems.

A philosophical observation: Much of the planning effort in algorithm design, and specifically in the design of data structures, is an attempt to take advantage of regularity (in some cases to force regularity) in order to simplify the description of data structures and processes. Example: if the elements of a domain can be ordered in an array or tree, it is easy to construct a name to element mapping algorithm (a "naming rule") for the elements of the domain, thus avoiding the necessity of providing individual names for all the individual elements in the domain. Another example: where processes can be described in iterative or recursive form, the specification of the processing to be done is thereby made simpler than if the entire processing actions had to be written out

sequentially. It is obvious that the more uniform, or regular, the processing is, the simpler is the processing algorithm. Much effort therefore goes into trying to make the processing more regular, even to the point where it is a little unnatural. One looks for the common actions in a series of processing steps to be done, puts these in an iterative loop, perhaps with tests and alternate branches to take care of the non-regular parts of the processing.

This search for simplicity through regularity goes on continually, particularly in the structuring of data. Attempts are consciously or unconsciously made to reformulate the data into composite elements having the same structure.

It is this strong desire for the simplicity which comes with regularity, I think, which has led programming languages to adopt the "regular" data elements—such as list, and set, and even binary tree—while tending to neglect the more complex elements such as those involving multiple relations.

Too often a programmer must mentally convert the concepts he wants to write into concepts which can be expressed in the language he has chosen to write in. This conversion too frequently consumes a large amount of his total effort. For example, assume it is natural for him to think of his data as numbers arranged in a tree and his chosen language is FORTRAN. In order to be able to express his ideas in FORTRAN language, he must first convert his data into rectangular arrays and convert the manipulations he wishes to make on trees into the corresponding manipulations on rectangular arrays. As a second example, assume that he wishes to process alphabetic data in ALGOL. Since ALGOL cannot speak of alphabetic characters, they must be converted to integers and processed as integers. When these conversion problems become too aggravating, the programmer may be impelled to choose another programming language. (Or to design yet another programming language. It is my strong conviction that this phenomenon is largely responsible for the current proliferation of languages.) An especially insidious effect of such restrictions is that they stifle fresh approaches to programming. With some restrictions removed on what can be expressed, a programmer should be able to devise algorithms that are fundamentally more efficient (that is, take fewer steps).

It is appropriate here to digress briefly to emphasize a fact which is widely known among computer people but often overlooked or forgotten: All programming languages for general-purpose digital computer's are universal, in that any computation procedure can be written in any such language, input-output operations excepted. (Among the cases where this idea of universality might be questioned is one where the character set available in a language was deficient; transliteration of the data would be required.) Thus, all computer languages (past, present, and, as far as anyone knows, future) do not differ in what they can accomplish, but they do differ in how readily some specific procedure can be expressed in them. Each language has been designed to make it relatively easy to express some class of procedures, to deal with some limited class of problems. To express in a given language procedures for which it was not designed is a relatively difficult task—sometimes miserably difficult—but never impossible.

A second major deficiency of essentially all current languages (there are a few languages for which I would say "moderate deficiency") is that they force the programmer, at the stage when he should only be concerned with working out the "logic" of his program, to be concerned also with the details of how the program will be implemented on a specific computer. For example, assume that the data to be processed is strings of alphabetic characters, that the computer to be used is a fixed-word-length machine without byte addressing, and that the language reflects this characteristic of storage. In such a case, the programmer must concern himself with how the strings are to be stored before he can define in detail the processing to be done on these strings. Should he store one character per machine word, which is wasteful of storage? Should he begin each string in a new word which will usually occur at the end of a string? Or should he ignore word boundaries, in which case he must address a string by both its storage location and character position within that location? A consequence of the presence of such implementation detail "woven" into a program is that it makes the program more difficult to change and therefore more difficult to check out. A further consequence is that such detail makes a program more difficult to understand and therefore more difficult to modify by someone other than the original programmer.

The third major deficiency of current languages is that they are "irregular" with respect to growth or change—in the sense that a simple change in a concept may necessitate a major change in the program. Simple examples of cases where this phenomenon can occur: increasing the precision of a computation, adding a column of data to a table. A more striking example is the extensive reworking necessary when a program grows to exceed the internal memory space available for it.

A fourth major deficiency of current languages is that they have a large number of conventions, established by the designer, which are implicit and inaccessible to alteration. By "implicit" I mean that the conventions are established in an instruction manual, but are not otherwise "visible" by inspection of the processor. By "inaccessible" I mean that the conventions are not represented in the processor in a way that can be accessed and modified by the user. Consider such a trivial example as a date. In processors which provide for a date, the format of the date is usually prescribed and unalterable. Such rigidity is both unnecessary and undesirable.

One reason for the "weakness" of current languages with respect to flexible data structures is that in most languages the structural relationships are left implicit rather than explicit. This limits the user to the relationship (or, possibly, a few relationships) fixed by the designer; the user is unable to construct his own, nor to test what relationships exist. In the present work, I have taken the position that all relationships must be explicit, and that they must be creatable, accessible (testable), and modifiable by the user. With the ability to create new relationships among arbitrary sets of data elements (each of arbitrary complexity) the user can create new composite data elements of arbitrary structure and complexity. With relationships explicitly expressed, relatively simple concepts can seemingly become quite messy (that is, complex, an unfortunate fact of life, perhaps). Since regular structures can be created readily by algorithm, this complexity is more apparent than real.

A fifth major deficiency prevalent in current languages is the necessity of specifying data structures at the time the program is written, with little or no opportunity to change these structures during program execution. We need to be able to build data structures as the need arises.

Standish was one of the first to recognize this goal and make a contribution to it (Standish, 1967). He has devised a method and a notation for the dynamic creation of new data structures. In the present work I have gone beyond what Standish did.

The continuing presence of the above-mentioned deficiencies is in many cases closely associated with a desire for efficiency. The designers of current languages have, with a few exceptions, an almost overwhelming desire to produce programs which operate "efficiently." I claim that these designers have sacrificed the productivity of the programmer in order to attain a questionable efficiency of object program execution. I further claim that the achievement of good object program efficiency is often an illusion because it is relative to the estimated time required by a program hand-coded in machine language to perform the same algorithm. The algorithm itself can be quite inefficient, however, due to the deficiencies of the language. Without these deficiencies quite possibly the algorithm could be a much more efficient one: that is, requiring many fewer steps. I therefore contend that to remedy the current deficiencies cited above, we must revise our attitude toward efficiency. What this revision entails will become clearer later.

AN APPROACH TO A SOLUTION

This section discusses the issues involved in trying to overcome the deficiencies cited earlier, and explains my approach to overcoming these deficiencies.

The conflict of flexibility with manageability. There is an obvious conflict between trying to get increased flexibility without increased complexity and, therefore, decreased manageability. Normally, the more "features" one adds to a programming language, the more conventions one has to learn and the greater the effort required to build a processor for the language.

Is there a way to get increased flexibility without increased complexity? I believe there is: by increased generality, and by the removal of unnecessary elements.

Generality is hopefully achieved by analyzing current languages to see what apparently separate concepts could be considered special cases of a more general concept. For example, perhaps we can find, or define,

"program statement" in a way which covers all the types of statements.

Some increase in flexibility will be achieved by the addition of concepts, which inevitably contributes to an increase of complexity. We compensate for the increase by removing some concepts from the composite picture of current programming languages. Not all concepts which are removed are really thrown away; rather they are "factored out" into another part of the programming task. Thus we are led to the idea of splitting the programming task.

Splitting the programming task. My approach to overcoming the deficiencies cited earlier is to separate the programming task into two parts:*

1. The working out of the "logical processes" of a program independently of the characteristics of any existing computer.
2. Specifying the details of how the program is to be run on an existing (hardware) machine.

For the purposes of this discussion we will say that the "logical processes" of a program, including the structural description of the data, are written in "logical language," and that the added information necessary to get such a program to run on an existing machine is expressed in "implementation language." Independence of the characteristics of existing computers means specifically the quality of being independent of the following characteristics of computer hardware:

1. Linear addressing schemes for memory cells
2. Fixed word sizes
3. Non-homogeneous memory units
4. Internal number representation
5. Serial processing (that is, instructions being executed in sequence)

No current programming language known to me is truly independent of all the above characteristics.

Another way of expressing independence: A program (in "logical language") which is completely independent of a real computer is one which would not have to be changed no matter what the hardware characteristics of the real computer might be. (Such a program can, of course, only be run on the real computer through the intermediary of another program; that is, a compiler or an interpreter).

That it is possible to make such a split of the programming task into two parts should be obvious. The logical language—the language in which the programmer accomplishes part 1 above—is by logical necessity a language for some hypothetical machine. It is a provable fact that any general-purpose computer can be programmed to simulate any other definable computer, input-output excepted. In particular, it is possible to write a simulator for the above-mentioned hypothetical machine and the logical language which it obeys. Every computer language processor is such a simulator. For example, an ALGOL compiler for computer X is a program which makes computer X behave like an ALGOL machine—one which responds to commands written in ALGOL language. Hence we see that it is possible for a programmer to write and debug a program using only the logical language—that is, without concern for how the program might be implemented on some specific hardware machine.

The second step—adapting the program written in logical language to run on some existing machine—is in fact mechanizable. Use of a simulator is one way of accomplishing this mechanization. However, mechanization of the implementation does not always yield an efficient object program. Hence the adaptation may in practice involve specifying some additional information via implementation language; for example, how the data is to be arranged in the available storage devices. This additional information is in the nature of "coaching" the processor of the language in order to produce an object program of satisfactory operating efficiency and reliability. In other words, implementation language will be needed for those things that we have not yet learned how to implement mechanically in a way that gives satisfactory operating efficiency. However, as computation becomes cheaper, as machines become faster, as larger high-speed memories become available, and as our ability to write processors improves, in many cases we should be able to dispense completely with implementation language.

The effect of this splitting of the programming task into two parts, each with its own language, opens the way to overcoming the deficiencies cited earlier. In considering the design of a logical language, the designer can then be free of the pressures imposed by the need for object program efficiency because that burden has been moved to the province of

the simulator and the implementation language. The way is thus cleared for designing logical languages having many more capabilities. By "many more capabilities" is meant "more convenient capabilities" rather than any increase in computing power. Among these capabilities should be greater ease of program modification.

Since the logical language is a language for an abstract machine, such a split seems to preclude its application to a certain class of programs we might call "computer-specific." A computer-specific program commands a specific machine to carry out actions peculiar to that machine. Examples of computer-specific programs are: a real-time computer program with radar inputs and teletype outputs, a monitor or supervisory program of an operating system, and a computer reliability check program. I think it will be the case, however, that computer-specific programs will, in contrast to "ordinary programs," be leaning heavily on the implementation language: that is, computer-specific programs will generally have only their skeletons expressed in logical langauge; implementation language will be required for the expression of details.

Inefficiency of program execution. There may be objections that such a proposed split will not permit a programmer to take proper account of hardware limitations, and hence program execution inefficiency will be a necessary consequence of splitting the programming task. That some inefficiency will result is probably true. It is my conjecture and strong belief, however, that no significant reduction can be made in computer programming effort and elapsed time unless hardware limitations are removed from the programmer's primary concern. The research reported here is, in a sense, the exploration of the consequences of ignoring hardware limitations when working in the logical language.

How much program execution inefficiency is likely to result from the proposed technique is a difficult question because efficiency is expressed as percent of some norm, or base. In this case, we don't have a satisfactory norm. Often "an expertly-hand-coded machine-language program" is used as a norm. There are at least 2 reasons why such a norm is unsatisfactory: (1) it embodies some unknown compromise between the antithetic goals of minimizing execution time and minimizing storage requirements, and (2) it generally implies very conventional programming techniques, yet we have no way of knowing whether radically different techniques might be

considerably better. I believe a better norm would be "an expertly-designed special-purpose machine." Compared to such a norm, an expertly-hand-coded machine-language program is grossly inefficient. I would estimate the efficiency expressed this way is in the neighborhood of 10%. This gross inefficiency is the price we must pay, at least at present, in return for being able to utilize general-purpose digital computers. Suppose for the moment that the proposed approach to developing a programming language resulted in programs which were 50% efficient compared to expert hand-coding in machine language. We could then say that these (the former) programs were 50% of 10% (that is, 5%) efficient compared to expertly-designed special-purpose machines. The idea that I am trying to stress is that heavy emphasis on program execution efficiency is rather pointless, at least where we are concerned with conventional present-day computers. To sum up, my whole approach is based on the premise that the possible resulting inefficiency of program execution will be a small price to pay if the efficiency and productivity of programmers can thereby be increased.

Some of the arguments against a broadly-applicable language have been eloquently summed up by Mitchell, Cheatham, et al ("A Basis for Core Language Design"):

1. The language processor must carry as overhead (in time and space) all of the mechanisms for language facilities which are not being used in a given program;
2. Each addition or elimination of a language feature is expensive;
3. Implementation of the processor on a small machine is likely to be impractical.

In this present investigation I frankly do not care about these arguments. I am trying to answer the question: "What would be the ideal language, if any, from the programmer's point of view?" However, when we become concerned with the problems of implementing any language that I might propose, these arguments become of concern. How, in fact, we must ask, can unneeded parts of a language and its processor be stripped off from a given program?

Emphasis of the present work. This splitting of the programming task leads to the development of two languages for programming. The first of

these languages is concerned with expressing a program for some hypothetical machine. The second of these languages is concerned with specifying the details of how the program is to be run on some existing real machine. This report deals almost exclusively with the logical language. The aim is to show that, given a willingness to tolerate a low efficiency of execution, we can make a significant step in the direction of improving programmer efficiency.

What I am concerned with is striving for a set of primitives (basic concepts) which is at a comfortable level of detail, and in terms of which all predictably useful concepts can be expressed without undue difficulty (ideally, with uniform degree of difficulty). By "comfortable level of detail" I mean something analogous to that available in current languages.

What I am attempting here is a formalization of many concepts which have heretofore been left to informal description in a programming manual, and to the intuition of the designer of the language processor. By "formalize" here I do not in general mean "put into mathematical-looking notation" but rather "make explicit by diagramming and test." One of my chief goals is to demonstrate that there is a gain to be had by making explicit many of the programming language concepts which have traditionally been left implicit.

REVIEW OF OTHER APPROACHES TO A SOLUTION

Of the current work in programming languages, very little is being carried out under a philosophy similar to the one I have advanced. Most developments can be categorized as single languages devoted to rather narrow classes of problems, with little or no attempt to separate the purely intellectual problems from the potentially mechanizable ones. Some languages, such as LISP, do try to minimize the programmer's concern with implementation; unfortunately these languages tend to be aimed at rather limited types of problems.

I have found only one hint of a development which attempts to separate the writing and debugging phases of programming from the achievement of an efficient implementation. This occur in a few sentences in the paper by E. W. Franks describing the programming system LUCID (Language Used to Communicate Information Systems Design)(see bibliography). No details are given, however, in this or any other document known to me.

There are 3 major current approaches to the problem of reducing programming effort:

1. The "generalized programming system," in which all of the functions needed for some class of problems are built into the programming system. An example is The MITRE Corporation's ADAM system (see papers by Burrows and Connors in bibliography). While in principle it may work, it has at least the following deficiencies:
   (a) It is aimed at a limited class of problems.
   (b) It is big and unwieldy, and complicated for the user to understand.
   (c) It has been designed and built around a specific machine complex (the IBM 7030 plus special peripheral devices); it seems to me that it will have very little "carryover" to the next generation of machines.
2. The "growing system" which accumulates all the programs and subroutines ever written for it and makes them all available for use as components of any new program to be written. An example, although not a good illustration, is the University of Pennsylvania's MULTILIST System (Prywes, 1963). Although a growing system could conceivably be used for a broad class of problems, it appears to have the deficiencies (b) and (c) just mentioned. Furthermore, its primitives (basic terms) are at a level of detail which is undesirably low.

   The "growing machine" as conceived and implemented by Professor Carr at the University of Pennsylvania is a flexible scheme for creating linkages among a set of programs through the intermediary of common pushdown lists (Ostrand, 1966). It provides for the definition of new data types as linear lists of other data types; the primitive data types are limited to the conventional ones: integer and real (as defined by the machine hardware), and identifier (expressed as a machine address). The language of command strings is basically a Polish suffix language (operators following operands). Because certain operators (such as QUOTE) require other than evaluation of its operands, and because the system conventionally evaluates its operands before it knows what

operators will be applied, these operators must be prefixed to their operands instead of suffixed.

3. The "something-for-everybody" language, in which the features of a variety of languages have been collected into one. The outstanding example of this type is PL/I (IBM, PL/I, 1965; Radin, 1965). PL/I incorporates the best parts of FORTRAN, ALGOL, COBOL, and a list-processing language. It attempts to avoid saddling the user with a mass of detail by being divided into nested sublanguages, so that a user does not need to know all about the language in order to be able to use a part of it. It may succeed in avoiding the widespread deficiencies of narrowness of application and language complexity. However, I do not believe it avoids two of the major common deficiencies discussed earlier: the one (deficiency #2) of forcing the programmer, at the stage where he should only be concerned with the details of how the program will be implemented on a specific computer, and the one (deficiency #3) of being irregular with respect to growth and change.

   Perhaps the greatest deficiency of PL/I is that it requires the user to keep in mind the internal representation of his data. One must know the internal representation in order to understand the rules of converting from one data type to another. One must know how the hardware limitations of the machine affect the data elements used.

The work of Tim Standish at Carnegie-Mellon University is not concerned with the overall problem of simplifying programming but it is an important contribution to achieving more flexible data structures. In his doctoral dissertation (A Data Definition Facility for Programming Languages," 1967), he has a powerful scheme for defining classes of objects, where an object is some individual variable or some list of objects. The definitions of data structures are themselves data structures which can be manipulated by program. His elementary types of objects are: real, integer, string, Boolean, and identifier. Overall his scheme is quite elegant and flexible. Provision does not exist for working readily with individual one-of-a-kind objects: such an object must instead be regarded as a member of a class having only one member. Provision is lacking for

defining new elementary data types.

These examples are mentioned, not to deride the work cited, but to point out that current approaches have deficiencies which I believe can be overcome by the approach described in this report.

MAJOR DESIRABLE CHARACTERISTICS OF A PROGRAMMING LANGUAGE

This section discusses the desirable characteristics of a programming language, given the goal of maximizing the user's efficiency. At this point the reader may not be convinced. He must take it on faith, or reserve judgment. One of the purposes of this report is to justify this choice of characteristics.

Generality of data elements and structures. In the structuring of data there are two opposing forces at work:

1. Economy of expression, and possibly also economy of storage, just because it is a natural human tendency to seek such economies.
2. Explicitness of the structure, so that it can readily be perceived and readily altered.

When economy is stressed, the explicitness of the structure essentially disappears. The structure is described on paper somewhere, but not explicitly stored. The user must learn and remember the structural information. It cannot be changed during program execution, and usually cannot be changed at all within the given language. When explicitness is stressed, economy must thereby be sacrificed.

Current programming languages have tended strongly to take the first course, economy, with the consequence of inflexibility. For problems well-structured in advance, this presents no real handicap. For problems not so structured, however, current languages have been somewhat cumbersome to use, because of their lack of flexibility in data structuring. This report argues, however, that the well-structured problems are now being handled by languages which have fixed and limited data structure capability. For each class of data structures we seem to need a new language. This is my explanation for the proliferation of higher-level programming languages. What is needed is a language with great flexibility in the data structures it can deal with. This paper proposes to show how such generality might be accomplished. In particular, for complete flexibility in data structures, it must be possible to express an arbitrarily- chosen relationship between two data elements, or among any set of data elements. Such a

relationship creates a composite data element, and this composite may in turn be an element of another composite.

The exact choice of a data structure depends on the way the programmer thinks and on the way he intends to manipulate the data. Our concern should be that he not be unduly restricted in the way he can express his data. In particular, we want to avoid the failing of many languages: namely, that a particular structure can be expressed in exactly one way—that there is a "standard way" of doing a particular thing.

Often a concept can be expressed in more than one way. Examples are:

1. A set can be expressed by enumeration of its members, or by an algorithm which generates the enumeration.
2. To each list (ordered set) there corresponds an unordered set.
3. For each list of characters there is a corresponding string of characters.
4. A regular structure, such as a tree with n branches from each node, is readily expressible in the form of a table.

Each method of expression may have its own advantages. Certain kinds of manipulations, however, can be performed only on one of several alternative modes of expression.

A related fact is that one of the most important features of a data processing system is the provision for reorganizing the data into a form most convenient for the job at hand. That is, we need to be able to transform data from one structure to another conceptually equivalent one. We contend that there is no definition of conceptually equivalent that fits all cases; hence, what "conceptually equivalent" means is up to the user to define. Except for some well-defined equivalences, "equivalence is in the [mind's] eye of the beholder."

The fact that a concept can be expressed in more than one way forces us to choose between two philosophies of processor design:

1. Require that the user be aware of a mode of expression so that he can decide whether a given manipulation is applicable.
2. Require the processor to be able to convert data to an equivalent form whenever needed to make it amenable to specified manipulation.

I reject alternative 2 as impractical at the present stage of development of processors. In this study I take the viewpoint that it is up to the

user to be continually aware of the form (structure) of the data that he is manipulating, and that it is up to him to convert to an "equivalent" form when necessary to make it more amenable to some specified manipulation.

I make the observation that the design of many programming languages is based on reducing sets of equivalent forms to single representations of each set. This has the advantage of simplifying the operations of the language, by reducing the variety of forms which can occur, but it destroys the convenience that alternative forms give us, and sometimes forces us to mold our ideas in forms that are inconvenient if not downright cumbersome.

Multidimensional data. The real world isn't always describable conveniently by linear (one-dimensional) strings. Two-, and even three- and higher-, dimensional problems need treatment by computers. The advent of computer-controlled graphic displays calls for two- and higher- dimensional languages. Multi-dimensional languages are those which not only deal with multi-dimensional data but also have capability for multi-dimensional expressions of commands, such as are represented in mathematical formulas (having sigma signs, superscripts, subscripts, etc.).

The difficulty with such multi-dimensional languages is that we know practically nothing about grammars for languages which contain other than linear expressions. What we do in practice is to find some way to map (set up correspondences of) multi-dimensional concepts into linear forms. Cheatham (1066) uses this technique to establish a correspondence between a desired two-dimensional notation and a one-dimensional representation of that notation. Put abstractly, his one-dimensional representation is a cartesian product of the possible two-dimensional elements, taken in a specified sequence. This technique could be extended to accommodate any two-dimensional language, such as the language of engineering drawings, the only real problem being the specification of the sequence in which the two-dimensional language elements are to be taken.

In keeping with this observation, we find that, in our discussions of how to treat data elements which we think of in terms of graphical structures, we have established correspondence rules for mapping between graphical structures and linear notation. For example, a pair of nodes with a relationship between them is mapped into an n-tuple containing the

node names and the relation name.

Generality in the sense of freedom. A programming language should be essentially unrestricted in what can readily be expressed in it. When thought of only in terms of the variety of procedures which can be expressed in it, this "feature" seems to be a banal triviality, since current languages seem to have this variety. However, when the emphasis is on the variety of data types and data structures, it is no longer a triviality. Clearly we are seeking a high degree of generality. But generality usually entails some penalty, such as less efficiency in some aspect, greater learning difficulty, or greater opportunity for error.

Universality. It cannot be emphasized too strongly that I am not claiming for my language approach "more universality" in the computability sense than is afforded by other current languages. All general-purpose languages can compute "some representation" of any computable entity. The proviso "some representation" is necessary to cover the cases where the input-output character set does not contain a needed character.

What is in fact important is the range and richness of the representations which the language offers. No language can offer the full range of possible representations, for that range is unbounded. Hence, the mental concept of a given algorithm must be mentally mapped into a corresponding concept in the available language. It is the difficulty of this mapping which concerns me. For example, consider that we wish to compute with a 3-valued variable with arbitrary names "T", "F", and "U". If the language has no provision for defining such a variable, then it must be mapped into something which the language does have. If it has integers, we could map T, F, and U into the integers 0, 1, and 2, and then define certain functions on these integers which would represent the functions we originally conceived on T, F, and U.

Another example is that of representing trees and operations on trees in a language such as FORTRAN which admits only the structure "rectangular array". A tree can with a non-trivial effort be mapped into an array. The array manipulations which correspond tree (list-structure) processing manipulations are complicated and unnatural.

Balance. A programming language should be balanced, in that it should be approximately uniformly easy to express any program of the types now

extant in the current languages or of the types suggested by the concepts to be listed in the next section. For example, it should not be the case that it is very easy to talk of homogeneous arrays but require a circumlocution (e.g., parallel tables in ALGOL) to talk about a non-homogeneous array. What constitutes a uniform basic set of language concepts is to a large extent a matter of taste and judgment.

Minimal discontinuities. An objective constantly in my mind has been that of making a language which readily accommodates change. I want a language which does not have radical discontinuities. This does not mean that every change in a program specification should result in a trivial change in the corresponding program. Rather it should mean, in my opinion, that a given class of changes (e.g., precision of computations) should not result in radically different amounts of effort to modify the program accordingly, depending on the values of the parameters of the change. To be more specific, in the case of precision, it should require approximately the same amount of user effort to alter a precision from one value to any other value. Notice that this goal is definitely not met by most current languages, because they are based on a fixed word length of the computer hardware.

Simple changes in concept should result in simple changes to a program. Examples of "simple changes" are: changing the collating sequence, increasing precision of a calculation, inserting a column in a table, change of domain size (of which increased precision is a special case), change of domain (e.g., from real to complex), change of domain units, introducing instances of a model which deviate from the model.

Elsewhere I point out that these types of changes are not limited to the pre-execution phase but can be made during execution.

Simplicity through emphasis on fundamental concepts. What I am developing is a basic framework within which can be determined without too much trouble a broader range of programming concepts than is available in current programming languages taken individually. I have not defined as many ready-made concepts as are in some current languages, such as some of the complex searches that are available in SNOBOL and FORMULA ALGOL. Hopefully what I have done is to show what basic concepts are needed in order to be able to define as needed a greater variety of concepts.

Hardware independence. A programming language should be unencumbered by considerations of the hardware of the computer and of the processor of the language (the simulator of the hypothetical machine). Specifically, as mentioned before, it should be independent of the following characteristics of computer hardware:

1. Linear addressing schemes for memory cells
2. Fixed word sizes
3. Non-homogeneous memory units
4. Internal number representation
5. Serial processing (that is, it should be possible to specify asynchronous processes)

These hardware-dependent characteristics can be "hidden" from the programmer by the simulator. It would be desirable to include in this list the input and output character set and the specific input-output devices. However, these latter cannot be satisfactorily hidden by the simulator. The language should be independent of processor considerations in the following ways: (a) It should not be limited by concern for the efficiency of the processor. (b) It should not require that everything be completely defined before anything can be processed; that is, it should be possible to write and debug fragments of programs. (c) A program should not be prevented from modifying itself in an arbitrary way; this includes modifying definitions as well as commands.

Dynamic modification of program, data elements and structures. There should be as little restriction as possible as to what can be modified during execution. In particular, we may want to increase the precision of a calculation in order to keep the precision of the result within bounds, to add elements to a domain, to alter a collating sequence.

One of the obvious consequences of choosing to have dynamic modification is that a program cannot be compiled—it must be interpreted. It should be clear that in order to be able to dynamically modify a program—that is, to be able to have a program modify itself—the program must be viewable as a data structure.

Program sequencing. We need a more generalized concept of program sequencing than is available in some conventional languages. We need first: parallel, asynchronous processing. Second, the ability to have

statements which are executed when certain conditions become "true" (called "chronic" statements by some). And third, we need more explicit control over branching, as to whether the branch is permanent, or temporary (meaning return later to the point of call), with the ability to remember, on a pushdown, the set of nested calls, and the ability to manipulate this pushdown.

Undefined functions and "garbage." Consider the class of machine operations which produce a result. A result is only valid if the operation ("function") is defined for the values of the arguments given to it. If it is not defined for its arguments, it still produces a result, but this result is "garbage", that is, meaningless. Most machines are constructed, however, so that garbage cannot be distinguished from valid results simply by inspection. In many applications, it is critically important that garbage values must not be permitted to masquerade as valid results. In applications such as space flight control or atomic reactor control, the processing of garbage values in place of valid ones can have disastrous consequences. It is practically impossible to achieve assurance that garbage values will not arise in a large computer program, for the simple reason that it is impractical to completely check out such a program.

The problem of avoiding undetected garbage can be dealt with if every function can be arranged to yield a recognizable "value" or "undefined" for those combinations for arguments for which it is not properly defined. The "overflow" signal generated in most computers is a very rudimentary embodiment of this concept, but it may go unrecognized if the programmer does not expect it and explicitly test for it. What is necessary is the existence of a "value" which is automatically recognized as "undefined" by every function. In conventional hardware such a value does not exist because every bit combination or digit combination has been assigned a numerical significance.

Two kinds of "undefinedness" should be recognized. It may be the case that the domain is defined but the member of the domain is not. Or it may be the case that the domain is not defined either.

It is not only in the case of evaluation of a function that a value of "undefined" can arise. It may occur when an attempt is made to reference a data element if the element does not exist; that is, if the reference

does not refer to an existing storage cell. It may occur when execution turns out to be impossible for reasons other than undefined parameters: e.g., a specified transformation may not be performable on a given (composite) data element, or a program statement may be syntactically or semantically incorrect.

Accessibility and traceability. The user must have access not only to the contents (values) of data elements, but also to all their related entities. Given any data element (atomic or composite) the user must be able to trace out (via the execution of program statements) all of the associations, or relationships, of which the given data element partakes. For each association thus determined, the user must be able to find out all the associated data elements and their roles in the associations. Examples: given a data element, find its identifier, or identifiers, if there is more than one. Given an atomic data element, or, more properly, a slot which can contain a value, find the explicit indication of what domain that value must be taken from. Given a data element which is a member of a set, find the data elements which are the other members of the set.

The traceability feature described here is a conceptual one. In particular it must not demand of the user any knowledge of how data is stored in a real machine.

Definitions and declarations. One of the needs of a programming language is to have explicit the consequences of definitions. The more explicit these can be, the easier to learn and to remember what definitions are in effect at any given time.

In some current languages, such as PL/I, there is a good deal of "learning" necessary in order to know the effect and scope of definitions ("declarations"). One must learn, for example, when the definitions become effective: i.e., whether they become effective at compile time or at execute time, and at what point of execute time. One must also learn at what point definitions cease to be effective: this generally occurs at execute time upon leaving a block under some specified set of circumstances. Ideally, there should be a simpler set of conventions.

In addition, one needs the flexibility of being able to alter definitions dynamically. That is, there should be a minimum number of definitions which are fixed and unalterable.

Removal of distinction between processing "phases." I submit that making a distinction between times that parts of a program are "interpreted" or at which variables may be "bound" is an unnecessary complication. These various "times" are: preprocessor (macro-processor) time, compile time, load time, and execute time. Distinction among these times and about what operations take place at which time is motivated by the desire to be "efficient". It is a feature I choose to do away with. The first reason is because of the unnecessary complication which is otherwise introduced into the language. The second reason is that since I wish to have dynamic modification of both processor and program, compiling in the conventional sense is not possible anyway, since such modification is prohibited by the nature of the compiling process. A "pure" compiler leaves no source language to be interpreted at execution time. Compiling is based on the static analysis of source language statements before execution. Such analysis is technically impossible to complete for a program which modifies itself in an arbitrary way. It is, of course, possible, for a program which modifies itself in a predictable way, through indexing, for example.

Provision for exceptions to general rules. An exceedingly important concept is that of being able to give general rules (algorithms and definitions), supplemented by special, and possibly contradictory, rules to take care of certain cases. When such rules are employed, the "exception rules" must be distinguishable, so that the processor will know that they take precedence over a general rule whenever the exception rule is applicable. What does all this really mean? What kinds of rules could we be talking about? The applicability of algorithms is decided by control sequencing mechanisms, and is wholly determinate. How about definitions? The applicability of definitions is dynamically determined, like the execution of algorithms. The order in which conditional statements, (algorithms or definitions) are tested automatically establishes the priority with which they will be applied.

Shorthand notations. One of the research goals is to provide as much flexibility of expression and breadth of concept as possible. Generality and economy of expression, however, are in conflict, hence we need the capability to introduce shorthand notations for frequently-used expressions. We need only provide the shorthand mechanism and let

the user introduce his own shorthand expressions.

Metadata. As important as being able to combine data elements to make composite data elements is the ability to associate explicitly with a data element a second data element which represents data "about" the first data element. This second data element we might term a "metadata element". Examples of such metadata elements are: an identifier, a domain "prescriptor" which specifies from what domain the values of the first element must be taken, an access code which limits the conditions under which the first data element can be accessed.

An alterable, prunable processor. How can we provide a wide variety of features, without: a) making the language processor too big and cumberson, b) making the user learn a lot of information he doesn't need, c) unnecessarily restricting the user with lots of conventions about the use of the features which are provided. Simply providing a "kernel" language, out of which the user can build anything he likes, is not enough; it is too much work for him to build all his own "tools". Providing him with a large library of subroutine packages is not enough; he must wade through catalog descriptions to find out if what he needs exists; then he may be frustrated to find that he would like just a slight modification of a cataloged program. Providing a large processor with lots of features is not enough; some features may be just excess baggage. It may happen that the user wants to modify some processor convention.

A respectable language should provide for him a great many of these useful definitions, such as formats of dates and times. The point is that these definitions must be accessible to him, in case he would like to make some additions or changes, so that he is not stuck with one set of conventions. In other words, the user should be able to get at and modify as much of the definitional mechanism as possible.

Envision a system in which we have a processor and an explicit set of rules which guide the processor. Among these rules will be the interpretation rules to be applied to data and imperatives. The user can, if he chooses, alter the rules to suit himself, at the risk of making an error. He can delete the unused rules, to make less "baggage" and possibly to speed up the processor. A safer elimination procedure would be to have the processor mark all the rules it used. After an

extensive checkout run the unmarked rules could then be deleted, although there will always be the risk of deleting a needed rule.

Toward meeting all these needs at once, I propose a "large" processor which includes a program library. The processor will largely be "accessible" to the user. He can prune off the parts he doesn't need. He can alter some of the conventions to suit his needs, not only before but also during execution.

One of the tricks is how to make the processor alterable and prunable. Among other things the interpretation rules used by the processor must be accessible by the user as data.

EXPRESSING AND PROCESSING DEFINITIONS

Types of definitions. One of my goals is to make more explicit and more simple the handling of definitions.

First, the concept of "definition" itself needs some discussion and clarification. "Definition" in the programming context means a variety of things ("Declaration" below is used in the ALGOL or PL/I sense):

1. Defining a "domain", specifying a set of "values" which a specified variable must take on. (For example, "Boolean" by convention specifies the domain (T,F).) Defining a domain is neither an "action", in the usual sense of imperative, nor is it an allowable substitution action to be invoked at some appropriate time. It is rather a specification that restricts some future action.
2. Declaration of a data element, possibly giving an initial value.
3. Declaration or definition of a composite data element, a data structure.
4. Declaration of a procedure.
5. A "standing order", sometimes called a "triggered statement" or a "chronic statement", to be executed whenever some specified condition arises.
6. An equivalence between two members of a data domain or between members of two different domains.
7. A definition of an ordering, such as collating sequence.
8. "Scope" of an identifier, usually defined by a means outside the language.
9. Shorthand notation (procedure call, macro) for a larger expression implying that some form of uninterpreted string substitution is

to take place.

Dynamic versus static interpretation of definitions. One of the philosophical problems which must be resolved early is the matter of when a definition becomes effective. Should it become effective at the time that it is made (that is, at the time it is written down or created during program execution), or should it become effective at the time it is used (interpreted)? This issue can be illustrated in the following way: Let us say that the definition B is made in terms of some other definition A: for example, a definition of a new domain in terms of one or more domains defined earlier. The question is: should a change in the definition A automatically invoke a change in the definition B? The user may have a legitimate desire to have it one way or the other. Therefore, we must have some set of rules by which he can control which is the case.

Since we are assuming a fully interpretive processor, we could if we wished assume the dynamic situation wherein a change in A results in a change in B. If we wished to avoid such a change, we can copy A and rename the copy, then change the copy. Since we want and have the dynamic change ability, this interpretation is preferable. In other words, under the dynamic interpretation, we can construct a static situation if we want it. Under the static interpretation, we cannot as readily construct the dynamic one.

The command interpretation of definitions. We need to identify a common thread in the above list of examples of definitions. That common thread is that all such definitions can be interpreted as commands which affect the contents of storage. Some of these commands will affect unseen storage within the processor: that is, they will in some way (which is not defined for the user) affect the action of the processor. It seems easier to understand and remember what each definition does if it can be thought of in terms of its action within the program or processor, even though this command interpretation of definitions seems somewhat contrary to intuition.

This command interpretation of definitions helps to avoid the complications of having to learn when definitions become effective and when they cease to become effective. If we take the command interpretation point of view, it follows that definitions become effective when they are executed and that they cease to be effective when dynamically superseded

by another definition. Furthermore, this command interpretation is nicely compatible with the need for dynamic rather than static interpretation of definitions, discussed above.

To summarize: definitions are to be established and interpreted dynamically. Note that data is automatically treated this way, so this choice follows naturally from our desire to be able to treat definitions as data until the instant of interpretation.

A vital consequence of this fact is that compilers are ruled out as possible processors for the language. A compiler necessarily treats all definitions as static, fixed at the time of compiling, which is before any execution has taken place.

NATURE OF A PROGRAM PROCESSOR

Basic to the idea of a programming language is the concept of a processor. A processor is a mechanism, possibly a hypothetical one rather than one realized in hardware, which translates into some appropriate action the expression written in the programming language. We call this translation "interpreting a program". Earlier, I presented the idea that definitions were interpretable as commands, hence we can regard a program as consisting of expressions, some of which is interpreted as commands and the rest of which is interpreted as data (operands). Which interpretation a given expression gets depends on the control mechanism of the processor. As a consequence, it is perfectly possible for a given expression to be interpreted at one instant as a command and at another as an operand. It naturally follows that a data element which is a program statement (a "transformation") can be created, modified, moved, etc., just as can an "ordinary" data element. The sequence in which data elements (necessarily having a certain structure) get interpreted as commands depends on the conventions for control sequencing, and on the manipulations performed on the control mechanism by other commands (such as "jumps"). As explained earlier, we want the processor not only to be able to handle a "single thread of control" but also to be able to execute multiple control "paths" (through a program) in parallel and asynchronously. In addition it must provide for the execution of "chronic" (or "triggered") statements which are commands to be executed whenever specified conditions prevail.

It is convenient to think of the processor as consisting of two main parts, a "command interpreter" and a "data interpreter". With this in

mind, we can say that whether a given expression is a program statement or a data element depends on which interpreter it is sent to (by the "control mechanism").

It is this data interpreter which is the heart of my concept of data elements. It is the task of the data interpreter to create, examine, and provide the means for modifying data elements. It can be said that the data elements and structure of the language (or any language) is simply the illusion that this interpreter presents to the user. The senses of "create", "examine", and "modify" have to be spelled out in greater detail. It is crucially important to realize that the basic actions of the data interpreter determine the whole character of the programming language.

We regard every program statement as expressible as a "transformation call". I have in mind a canonical form for expressing these transformation calls: it is the familiar "prefix form", of which an example is:

T(A,B,C);

Every statement in any programming language should have a counterpart in this canonical form. For each statement type in each programming language there might have to be a conversion rule for converting it to the canonical form. In some cases, some "understood" information might also have to be supplied during this conversation.

With this viewpoint just outlined, the loading and execution of a program consists of the following two simple steps:

1. Read in and store the following data elements and structures, some of which will later be interpreted as program statements.
2. Begin by interpreting the data element named "P" as the first program statement.

The idea that there is a canonical form for all transformation makes the interpreter mechanism a conceptually simple one. The difficulty of any such interpreter is that part which must examine (parse) a statement and in effect reduce it to canonical form. If we could agree to keep statements always in canonical form, the parsing mechanism would indeed be simple.

LANGUAGE DEFINITION

Defining a language. Every concept in a programming language is either expressed in terms of other concepts in the language or in terms

of concepts which are "primitive", not further defined in the language. These primitive concepts are defined rather by the behavior of the processor, and the processor itself is expressed, at least partially, by concepts which are not defined in the language which it is designed to process. (This assertion is not based on personal opinion but rather on some fundamental ideas from the field of formal logic).

Extending a language. It is appropriate to point out here that any claim that a language is "extendable" must be understood with the above explanation in mind. Any extension, in any language, is made in terms of the concepts for which definitions already exist, and in terms of primitives. It is clear that the possible extentions are limited to what can be expressed by legitimate (that is, defined) compositions of the already defined terms. The definition of definition requires that by repeated substitution of definitions in place of the defined terms, ultimately all defined terms are removed from an expression, thereby reducing the expression to one composed entirely of primitives. Thus it is evident that user-created definitions can only introduce convenient "shorthand" notations. The important consequence of all this is that a language's primitives and syntax limit what can be introduced by user-created definitions.

Self description. It is sometimes claimed that a language can be "self describing", or that a processor can be written in its own language. This is a misleading claim. Some parts of a language or processor may be describable in terms of the other parts, but this cannot be true for the whole language or processor.

In particular, much as we would wish it to be the case, a data element cannot be self-describing. When we look at some entity, we cannot interpret it unless we have been given an interpretation rule associated with it. For example, the string of characters

13

"means" a variety of numbers depending on the number base (the interpretation rule) that is understood. Suppose that instead of having the number base understood, we write it down explicitly, so that now we have the number pair

13 8

that is, "thirteen to the base 8". Is this now self-describing: interpretable

without recourse to an interpretation rule? No indeed. The interpretation rule for 8 is understood; among other things we understand it to be written in base 10!

The conclusion that we draw is: an entity may contain some parameters which pertain to the interpretation of the rest of the entity, but the interpretation of those parameters must be specified by some understood rule which has been previously-specified. In other words, no entity can be wholly self-describing in any formal sense.

The choice of primitives. Selecting what I think is an appropriate set of primitives for a programming language is one of the major objectives of this study. At this point I emphasize that in the choice of primitives I do not take the attitude of a typical mathematician. He is interested in elegance—the minimum number of primitive concepts in terms of which he can express all other concepts of interest to him. I am primarily concerned with simplicity and convenience. This leads us to some middle ground between two extremes:

1. A minimal set of primitives, sometimes resulting in complicated combinations of primitives to express intuitively simple ideas.
2. A large, and therefore inconvenient if not unmanageable, set of primitives.

I want to maximize the usefulness of the set of primitives to the user. The choice between the extremes is a matter of judgment and taste.

OTHER ASSUMPTIONS AND COMMENTS

This section contains some miscellaneous comments which conclude the setting of the stage for the research investigation.

Storage deallocation. Matters of storage deallocation are not considered relevant to the logical language. The argument for taking this point of view is that with enough storage available, storage deallocation during a given program will not be needed. Storage deallocation will therefore be relegated to the implementation language.

Scope of identifiers. A scope of an identifier of a data element is that dynamic period (during execution of a program) during which the identifier can be used to reference that given data element. In languages with nested block structure, such as ALGOL and PL/I, the scope of an association of an identifier with a given data element is usually for the period during which "control" is in the block or procedure in which the

data element was declared. This association may be temporarily superseded if in a block B contained in block A the same identifier is used in a declaration of another data element.

Being able to define scopes of identifiers serves two purposes:

1. It can be used to signal automatic storage allocation and deallocation.
2. It permits the repeated use of a given identifier for more than 1 data element, as long as, at any given instant, the identifier refers only to 1 data element. This is most useful for identifiers of "local variables" of a procedure (data elements declared within the procedure) and formal parameters (bound variables) of a procedure.

These uses of scope are conveniences, however, rather than necessities. Since in this investigation we are not concerned with storage deallocation, the storage deallocation function of "scope" is of no usefulness to us. It is not difficult to avoid multiple use of the same identifier, and there is no great hardship in requiring the user to keep his identifiers unique. He is already required to do this for labels (identifiers of program statements) at any given "level" of block structure, and to do this for data. Qualifications of identifiers is a standard technique applied to data. It is trivially simple to extend this technique to apply to labels in multiple levels of block structure. With the facility of dynamic modification and multiple parallel control paths, defining scopes is a difficult task. For the foregoing reasons I choose to ignore the matter of scope.

CHAPTER 3. OVERVIEW

This chapter is concerned with an overall look at the research developments in this report. It summarizes what is to come, and outlines the major ideas, some of which are new and some of which are simply resurrected and revived.

I hope that a major contribution of the research has been the clarification of a number of programming language concepts which have heretofore been explained very loosely and intuitively rather than rigorously. Examples of the concepts which have been explored are:

1. what kinds of treatment can parameters of a procedure undergo,
2. what kinds of relationships need to be provided for,
3. what are the various ways in which a data element can be referred to,
4. what are the implications about copying when invoking a procedure,
5. what is the nature of the result of executing a procedure which is a function (in the mathematical sense).

Perhaps one of the most valuable contributions of this work is to show the complexity involved in seemingly simple language concepts. The very simple concepts of formal logic and mathematics do not remain simple when they are carried over into the programming context. In the analysis carried out under this research project, these complexities have been brought to light and examined. In particular, in being able to see what flexibilities are possible one should be able to recognize more readily the inflexibilities of current languages, and have more insight into the reasons for these inflexibilities.

The kinds of actions possible in the system outlined here include being able to create identifiers and data elements, and to destroy same, to gather data elements into composite data elements, to manipulate identifiers as data [including altering spelling, making lists of same], to search the structure and contents of data elements, to create and to destroy arbitrary relationships between data elements, to be able to regard strings either as integral units or as composites of characters. Current languages in general do not have these kinds of facility; of course there are occasional exceptions.

Aspects of the "logical" part of a language. A logical algorithmic language (independent of implementation on a specific computer) can be viewed as having three major parts: data elements, data transformations, and sequencing rules. In some languages the statements to be executed can be manipulated as data: we take this capability as the more general case and accordingly regard program statements as having the same structure as "ordinary data" until the instant that a statement reaches control, to be interpreted as a command.

A data element can be either simple or composite. A composite data element is composed of other data elements with some specified relationships between them. The character of a programming language is completely determined by its primitive (basic) data elements and the allowable ways of combining those primitives into composite data elements.

Data transformations can be divided into two classes: (1) those which determine a value, (evaluate a function, perform a mapping) and (2) those which create or modify a data element. As I mentioned earlier, destroying a data element is only necessary for the sake of using storage efficiently, and hence belongs to an "implementation language", not to the logical language.

Sequencing rules specify the dynamic sequence in which data elements are to be taken and interpreted as commands. Such rules specify: (a) start and stop points, (b) changes of sequence, either temporary or permanent, from the static sequence in which the statements are stored, (c) execution conditional upon a predicate, where the predicate may be evaluated either when it is encountered or whenever any of its operands acquires a new value, (d) iteration of a set of statements over a set or sequence of parameters, (e) initiation of parallel execution of a specified set of statement sequences.

A language for a hypothetical machine. This report describes a language for just such a hypothetical machine as mentioned in the Introduction. The main characteristics of the language are:

1. Storage is viewed as an unlimited supply of "slots", or "cells". Each such cell can contain a "value", which is some string of characters. Association links of various types can express relationships between pairs of cells, or among sets (ordered or

unordered) of cells. Associated with each cell containing what we think of as an ordinary variable can be another cell which specifies the domain of the value in the former cell; in other words, the domain is the name of a rule for interpreting the value. An association or relationship among cells creates a composite data element. A given cell can participate in multiple relationships.

2. All statements (commands) are in a standard form: the name of the command, or action, followed by a list of parameters. The parameters may themselves be transformations, and so on, to any depth of nesting. Transformations which appear as parameters are usually functions: transformations which return a "result". This result, however, need not be a simple value; it can be a composite data element (which may be arbitrarily complex).
3. A program is a data structure (a composite data element) which is a network of statements, not necessarily connected. (That is, there may be more than one separate network.) A program is executed by having "control" trace through such networks, and executing statements in the sequence indicated by the network connections, except when "jump" statements are encountered. If a program consists of more than one such network, each network is executed independently (simultaneously and asynchronously). Some of the networks may represent program segments to be executed only when certain conditions arise (alternatively expressed, "are true"). Provision exists for several types of "changes of control", or jumps: they may be permanent ("go out"), or temporary ("perform").

This hypothetical machine has not been tested, other than on paper. Among the recommendations given is the one that a simulator should be written and used, so that the concepts presented herein will receive some realistic testing of their usefulness.

The major ideas. The major ideas presented in this report are discussed below. Some of the ideas are new, at least in the sense that they have not been reported in the literature. The ones which are not new are hopefully a contribution to the state of the art by way of emphasizing matters that have been given insufficient recognition.

1. The key to obtaining a wide variety of data elements without increasing the complexity of a programming language is to provide generalized data elements. The most general structure is the general network, which provides for arbitrarily-specified relations between arbitrarily-specified elements. Too much generality, however, can be inconvenient. Ways are needed to specify more restricted structures within this general framework. The most important conceptual device for expressing composite data structures is the n-ary relation. The mathematical concepts of a relation is a set of ordered n-tuples (lists), which our generalized data element can express. The components of such n-tuples are frequently names (that is, strings interpreted as names), but can also be pointers (usually numerical addresses in conventional machines). With such a generalized element we can, for example, construct partial orderings, total orderings, and multiple orderings. The ability to express arbitrary relations between data elements gives us the capability of using relations weaker than ordering: such "directed association" permits, among other things, the use of circular ("ring") structures.

The composite data elements available in current languages encountered thus far all seem to be special cases of a set of ordered sets (lists) whose sizes (lengths) may differ and whose components may differ in type.

2. Domains of values of a "new" type of data element should be creatable within a programming language (during program execution). What is needed is the ability to name a new domain and define by enumeration or by algorithm arbitrary strings which represent the members, or values, of this domain. In conventional programming languages this is, in general, not possible. For example, we could define a three-valued domain with the name "3V" and with members "-1", "0", and "+1". We should be able to go further and define synonyms for any members. The example just given illustrates an elementary (non-composite) domain. However, the ability to define domains must include composite values. For purposes of explanation here, a composite value can be thought of as a vector of values; actually it is a much more complex object.

3. We should be able to define arbitrary functions on the members of any domains. Such functions will give a proper result only for proper (defined) arguments, and a result of "undefined" otherwise.

Here is an illustration of defining a domain and a function upon it: Let there be a new domain, called "Truth", containing the representations

"T", "F", and "U". Let there be a function, called "T.AND", defined on this domain by the following table:

arg1	arg2	result
T	T	T
T	F	F
T	U	U
F	T	F
F	F	F
F	U	F
U	T	U
U	U	U

where arg1, arg2, and result are all from the domain "Truth." We picked a particularly simple example. The individual arguments and the result of a function, of course, need not be from the same domain of values. The representations in a domain are not restricted to single characters; in general they will be strings. It will sometimes be impractical to define a function by means of a table (even though it is theoretically possible to do so); some functions, such as addition, will be defined by giving an algorithm which operates on strings representing values.

4. The concept of one element serving as the "name" of another element is in itself a relation. Algorithms deal in manipulations of elements. But in fact we never "see" these elements nor exhibit them. We refer to them only by their names or some directions for locating them. Even so familiar a mark as "101" is not a number, for example, but is rather a name of a number. It is not the name of a number, for the same number can have many names. Furthermore, this name can name many numbers, until the number base associated with this name is known. Entities which serve as names for other elements are sometimes to be treated as data elements. The name "101" may at one moment be the decimal name of a number, and at the next may be the name of a string whose second character is zero. To further complicate the matter, the name "101" may in turn have a name, say, "BVAL". Programming languages have tended to avoid these interpretational ambiguities by fixing the interpretations of names, and thus they have tended to prevent the manipulations of names as data and the use of indirect naming (allowing names to have names). If this flexibility is desired, then clearly the programmer must have control over which way an element is to be interpreted.

5. The explicit association with each data element of its "type" designation provides another degree of flexibility and another technique for discovery of programming errors. "Type" in the case of a simple element is the name of the domain from which the element's value is taken. An example of such a type designation is "integer". "Type " in the case of a composite element designates a class to which the composite belongs. The inclusion of a type indicator in each element can be used as a basis for testing the type of the element and making further action contingent upon its type, and also upon its structure. The explicit carrying of type information makes it possible to avoid the execution of a function or procedure when its parameters are not of the required type. For example, a procedure defined for three input parameters of types integer, string, and label, respectively, should not be executed if any of the input parameters are of the wrong type, but should instead return with a specific error message.

An example of type might be length-in-inches. Even though the representation of a specific length is the same as for an integer, a function defined for arguments from the domain of length-in-inches could be arranged to refuse an argument from the domain of length-in-feet.

6. The concept of having a value called "undefined" is a valuable one for the rapid discovery of programming errors. With proper use of the value "undefined", any program can readily be prevented from computing "garbage", values which look reasonable but which are in fact meaningless.

7. There can be two fundamentally different kinds of composite data elements (data structures). The first kind contains its own structural information; in the second kind at least some of the structural information is abstracted and put in some place common to a set of similar elements. In the second type, there can be defined a class of similar composite elements. Each specific occurrence of a composite element is then an "instance" of the class. Such a class description we have called a "model". (In the case of a simple element, we call the model a "type" and an occurrence a "token".) In the first kind of composite data element description, a composite element is created explicitly giving relationships which tie together elements already defined. In this latter case there is no model; we could call an occurrence of such a data element a "model-less instance". Most, if not all, current programming

languages provide for only one of these methods. This is a serious deficiency, for each method has its advantages.

8. The user must have complete control over data elements and structures. Having this capability implies several things. It implies that all available information about a given data element must be explicitly stored and accessible to the user. Not only values should be accessible and modifiable but also a variety of information about a data element: such as its structure, the specific relationships between its parts, restrictions on any of its parts as to values it may take or as to what kinds of access it can have, when it was last modified, etc. Having this capability implies that components of composite data elements should not only be accessible in the basis of an identifier but also by its position and by its relationship to another explicitly-known data element. It implies that the user is not limited to the set of relationships fixed by the designer, but can construct his own, and test what relationships exist. With the ability to create new relationships among arbitrary sets of data elements (each of arbitrary complexity) the user can create new composite data elements of arbitrary structure and complexity.

9. The user should have explicit and flexible control over the procedure call mechanism. This means he should have control over how the parameters are interpreted, and when they are interpreted. The conventional procedure call mechanisms do not in general provide such flexibility. ALGOL does give the user a choice as to whether a parameter is to be "evaluated" ("call by value") or substituted without evaluation ("call by name"); but it does not give the user any control over when this parameter treatment takes place: it is always done upon entrance to the called procedure. The user should be able to assess dynamically whether a parameter is defined, so that "garbage" cannot be passed as a parameter. The user should be able to examine the parameter "type", or "domain", and make the treatment of the parameter, and possibly the domain of the result, conditional upon the parameter type.

10. The concept of "string" has a dual aspect. On the one hand, a string is frequently used as an identifier or name; in this role, the string is an indivisible entity. On the other hand, a string may be an ordered set of characters; in this role, the string can be inspected, dissected, elements replaced, etc. The same string can be viewed in only one

of these ways at a given instant; it must be a data element with describable characteristics. One of these characteristics is whether the element is simple or composite.

The same thing may be wanted in both roles; thus provision is needed for converting from one form to the other. This duality of roles gives us the ability, for example, of composing a string and then using it as an identifier. A second example would be the ability to alter the spelling of an identifier.

Furthermore, a string may be wanted as a unit, a "character", in a "higher-level alphabet" of strings. This is a fundamental device for creating an unlimited number of symbols out of a limited number of characters.

11. It is useful and helpful to view definitions and declarations as commands. As discussed in more detail in Chapter 2, this viewpoint makes it easier to explain and easier to remember what effect definitions have and the times at which they become effective.

12. The provision of multiple, equivalent viewpoints of a given concept makes a significant difference in a programming language. It is frequently the case in programming languages that the user is constrained to look at a concept in only one way. This has the advantage that one user can know in advance how another user has viewed and treated the concept. It has the disadvantage that full flexibility needs the multiple viewpoints. By restricting the viewpoints to one, in any given instance, the user may lose some of his ability to express his ideas in the way most natural to him.

In the research work reported here, I have made a concerted effort to avoid limiting ourselves arbitrarily to single viewpoints. The consequence is that the user must pay more attention to which of several possible viewpoints is being used. He must know, for example, whether a given set is defined by an enumeration of its members, or by an algorithm which generates the members.

13. Such fundamental mathematical notions of "set", "relation", and "function" are not expressible simply and uniquely in the programming context. There are multiple, equivalent forms of expression, but the equivalence is not automatically invoked. It is impractical to legislate the use

of only one of a set of equivalent forms, because each form has its advantages. It is the burden of the user, therefore, to know which alternate equivalent form is being used in a given circumstance.

14. The ALGOL concept of "call by value" has been replaced by one of "call by reference". A parameter called by reference must either be an identifier of a data element or an expression which when "executed" or "treated" will yield an identifier of a data element. Where identifiers of data elements such as literals and function results are not supplied by the user, they must be automatically supplied by the processor.

CHAPTER 4. DATA ELEMENTS AND STRUCTURES

FUNDAMENTALS

"Basic elements." An algorithmic language is largely characterized by the kinds of data elements that can be described and manipulated. One of my goals is to define a set of elements that include the elements collectively available in current "higher-level" programming languages. I call this set my "basic set" of elements. "Basic" does not imply that the elements are not composites of other elements. Rather it connotes that it is the set of elements which are "given"—"undefined in the language system", as the logician would say.

Those basic elements which admit of no further subdivision, or alternatively stated, have no structure or are not composite, I term "simple data elements", or "simple elements". Rather obviously, those elements which are not simple must necessarily be composite. As we shall see later, the components of composite elements may themselves be composite elements, so that the definition of "composite" is necessarily recursive. Also as we shall see later, I will choose some frequently-used composite elements to be in the "basic" category.

Alphabet. An alphabet is some set of distinguishable marks which are considered not to be decomposable into other marks without some special operation. A language is based upon having a given alphabet. In the context of programming languages, the alphabet is some set of characters that can be concatenated into strings. Examples of an alphabet for programming are the set of characters on a typewriter keyboard, or the set on a card punch.

We might ask the question: Can the given alphabet for language be extended by means expressible within the language? It is indeed possible to conceive of a mechanical processor which can perceive a character, add it to the stock of characters in the processor's working alphabet, and be able to output the character in a form recognizable by humans. Such a processor would have to have some means of recording and reproducing an arbitrary mark. Most programming language processors (computers) available today do not have the extension capability just outlined. For this reason, I have chosen to limit the present phase of investigation to languages and processors with fixed alphabets.

I shall henceforth call the alphabet the "given alphabet" The given alphabet includes all the characters found on a standard typewriter keyboard, plus some other characters which will be specified later.

String. The notation of string is primitive (not defined in the language). A string is a sequence of any characters of the alphabet, possibly including blanks.

Having chosen the notion of string as primitive we are faced with a problem: how can a string be split into its component parts? We have on the one hand the notion of a string as a unit, playing the role of a single, individual symbol. On the other hand we at times want to consider a string as an ordered set of characters, and have the ability to scan and to modify this set. It is this dual role for strings which presents the problem. We simply note the problem at this point; its resolution will be treated later after the concept of composites has been developed.

Note that a string as here defined can contain quote marks. When a string containing quote marks is quoted—surrounded with quote marks—ambiguity may result. That is, it may not be possible by inspection to discover the original string. Techniques exist for preventing such ambiguity from being created; these are discussed elsewhere, in a section entitled "The control character interpretation problem".

Defining sets. One of the primary abilities needed in a programming language is that of defining sets. One class of members of sets is value-strings. Another class of members is data elements, both simple and composite.

There are a variety of ways of expressing sets. All these ways should be usable by the programmer:

1. enumeration, a listing of the names of the members. (If the members are value-strings, their names are simply the quoted value-strings.) Such an enumeration is represented as a composite data element (unordered).
2. generation, an algorithm (or formation rule) for generating the names of the members. For example, "all strings of length less than 10 which are composable with the substrings 'A', 'AB', and 'C'."
3. restriction of a given set by a condition (a predicate). Alternatively stated, this is a decision rule, a recognition algorithm, for deciding if a member of some given set (some "universe of discourse") is a member of the desired set. The desired set is in

effect derived by examining in turn every member of the desired set. Where the universe of discourse is small, this "total examination" scheme is workable and useful; where the universe of discourse is not small, however, this scheme is impractical. There are some interesting special cases of this method:

a. stating properties that members, assumed to be composites, have in common;

b. being in a class of data elements which stand in a given relationship to a given entity. Specific examples:

1) having a given element as a model;

2) being an identifier of a given element (a set of identifiers thus defined are synonymous).

(Some fairly cumberson devices have been used to express this concept. In AEDNET, for example, this set of relationships is expressed by tying the related members together in a "ring" structure.)

4. some <u>set-theoretic combination</u> (that is, union or intersection) of sets already defined.

Note that some sets exist by virtue of satisfying some predicate (e.g., have some specified relationship to a specified set of properties), and are not otherwise explicitly listed as being in a list or set.

For the purposes of testing whether a member is in a given set or list, the user theoretically need not be conscious of whether the answer is delivered by a recognition algorithm or a list search. For the purpose of modifying the set, however, the user must be conscious of the distinction, because the modification of each must be done differently.

From a practical standpoint, however, even the asking of questions about set membership would appear to require the user to know how the set is defined. To do otherwise would burden the processor with deducing how the set was defined; this seems to be a non-trivial task.

I make the provisional definition that "set" is a general, abstract concept not representable in a unique way. The construction, testing, and modification of sets demands that the user know, and be explicit about, how the sets are defined. The testing of set membership must then be spelled out explicitly according to the nature of the set. E.g., is X in an association list (X,Y,Z)? Does X stand in the relationship R to

element E? Does X have elements A,B standing in the relationship P to itself? Since a list is a set with a total ordering imposed in it in some way, these arguments apply to the concept of list as well, and, in general, to any ordering. In this connection, it may be useful and desirable to have transformations which will convert one type of set representation to another.

Sets. It will on occasion be desirable to provide for sets having repeated members. In order to be consistent with the well-established mathematical definition for "set" I call such an entity a "set with possible duplications" or, perhaps, abbreviate it to "set wpd" when I am lazy.

A set is a single entity whose cardinality is 1. We will on occasions encounter an aggregation which has not been explicitly defined as a set. It will be more convenient for certain purposes not to regard this as a set but rather as an entity having the cardinality of the number of members. I arbitrarily choose the new name "aggregate" for this entity.

Equivalent concepts. There are fundamentally different ways of expressing the same concept. In our thinking, and in expressing a concept in natural language, however, some means must be available for dealing with equivalent concepts. Of course, it would be desirable to have the processor recognize such equivalences so that the user did not have to concern himself with them. This is too much to expect at the present time, however; I dismiss this possibility from further consideration. As a consequence, we must place on the user the burden of knowing which of several equivalent forms he may be using, and of converting from one equivalent form to another when the need arises. Another way to phrase the problem is that a set of concepts may be equivalent at one level of detail (or "level of abstraction") but may not be equivalent at another level. The import of this philosophy will become clearer as the concepts of data elements are introduced and discussed.

Notational conventions for brackets. I adhere to the following use of brackets:

[.....]	imply an ordered set, or "list"
{.....}	imply an unordered set
(.....)	imply grouping, without specification as to ordering

DOMAINS, MEMBERS, AND VALUE-STRINGS

A domain is defined to be a set of entities where "entities" is about to be defined by example. Examples of the domains used in current languages are:

1. "real", meaning positive and negative rational numbers (up to some magnitude usually dictated by the word-length of the machine on which the processor is implemented);
2. "integer", meaning positive and negative whole numbers (up to some magnitude usually dictated by the word-length of the machine on which the processor is implemented);
3. "Boolean", meaning the set of 2 members: "true", and "false".
4. "String", meaning the set of all strings composable with the alphabet of characters available on some assumed processor, and with a specified bound on length.

The members of a domain are variously describable as "entities", "concepts", or "values". However they may be described, the individual members are ideas, and these ideas are not directly sensible by a machine. Each member is therefore represented by some string of characters. It is this string of characters that can be sensed and manipulated by a computer program. These strings of characters representing members of domains I call "value-strings" because they represent "values", because they are strings, and because they need to be distinguished from other types of strings.

To repeat, computer programs deal with classes of entities called "domains of value-strings". Value-strings and domains are discussed in more detail below. This research is based on the idea that the user should be able to create domains of whatever value-strings he wishes, and to be able to define functions on these domains.

Value-string. A "value-string" is a member of the set of finite, non-null strings* which can be formed from the given alphabet.

---

*"Finite, non-null strings" is the mathematician's way of saying "all strings except those composed of no characters and those composed of an infinite number of characters.

A value-string represents a value, where a value is a member of some set of concepts called a "domain". A processor cannot sense this member (this

value): it is only a concept, such as a "number", a "truth value", a "state" of an object, or a "value of a property". We can only represent a member of a domain of values, though possibly we can represent it in more than one way. It is only in terms of these representations, which are arbitrarily chosen, that machines can be made to deal with concepts. The "meaning" of a value-string may be suggested to us by its form, but its meaning or behavior in an algorithmic process is wholly determined by the functions that are defined on it.

Words sometimes used to mean what I define as "value-string" are: "value", "literal", "string", "representation", "constant", and "non-logical constant".

A value-string may be used to represent members in more than one specific set of concepts (domain of values). Or, what amounts to the same thing, a value-string can simultaneously be associated with more than one domain of values. In such cases, to avoid ambiguity, it is necessary when exhibiting a value-string to specify at the same time to which set or domain it belongs.

A "value" can then be defined as an abstract concept or invisible entity which is represented in some processor by a value-string. I use "abstract" and "invisible" to emphasize that the represented member is not present inside the processor: only its representation is present.

Domain of values. A "domain of values" is a set of concepts. A domain of values has a corresponding set or domain of value-strings, which is some subset of the finite, non-null strings composable with the given alphabet. An example of such a set of value-strings is {T,F}. (Note to the non-mathematician: the curly brackets are conventionally used to mean a "set", a collection of elements in no particular sequence)

A domain of values has a name (possibly more than one name), which itself is a value-string from the set of possible strings. I call such a name a "domain designator". An example would be "Boolean", which is the usual name for the domain whose associated value-strings are "T" and "F". It is a convenient and harmless ambiguity to let a single string serve both as a name for a domain of values and as a name for the set of associated value-strings. Henceforth, I allow this ambiguous role of the "domain designator". After this present section, however, my use of "domain designator" will refer consistently to sets of value-strings. A set of value-strings I will consistently call a "domain."

At first it might seem desirable to have all possible value-strings in one big domain—a "universal domain", or "universe of discourse". It is desirable, however, to divide the set of value-strings (that represent values) into named, possibly overlapping, subsets (domains) for the following reasons:

1. To permit the use of a given value-string to represent different concepts. Example: "2" could simultaneously represent a length in inches, a weight in pounds, an amount of money in cents, and the number of members in a specified set.
2. To make possible the prevention of nonsense: the prevention of the computation of "garbage". Examples of nonsense: (a) to set the value of a Boolean variable to "30"; (b) to set the value of a data element intended to take the values of weight in pounds to the character "Z"; (c) to multiply inadvertently 3 feet by 18 inches expecting to get an answer of area in square inches.

We need the ability to specify and name new domains to suit special purposes, for reasons of convenience and checking, and to get an output in a specified form. It should be possible to define new domains not only at the time a program is written, but also during the execution of a program.

Defining a domain. A set of value-strings corresponding to some domain of values can be described in any of several basic ways: by enumeration, by a generation algorithm, or by a universe of discourse and a recognition algorithm. These basic techniques for describing a set can be used in combination to describe other sets. It is also possible to create new sets out of set-theoretic combinations of sets already defined.

Enumerated domains can be in the form of composite data elements, while domains described by algorithm will be implicit in those algorithms. In order to be able to construct a new domain, however, the concept of domain must already exist, as must the domain of strings. The concept of domain is created by having as part of the given system a domain of domain names. This is a composite of fixed structure, telling for each domain whether it is defined by a data element (a composite) or by an algorithm, and for the algorithm case it may give an identifier of a generator of the members of the domain, an identifier of an existence recognizer for members of the domain, and the identifier of an equivalence

generator and recognizer. An initial entry in the domain of domains must be the name STRING. It should go without saying that the given system must also have the functions and transformations needed to carry out this domain definition process, including the algorithms for generating and recognizing strings.

It will on occasion be desirable to be able to specify an ordering of the members of a domain. For example, for the domain of characters, one might wish to be able to define a total ordering of the value-strings: this ordering is commonly referred to as "collating sequence." Another example: to give in order the designations of the hours in a day: the first hour is anomalously numbered 12 rather than 0, so we have the ordered set [12,1,2,3,4,5,6,7,8,9,10,11]. (The reader is reminded that throughout this work, I shall consistently use the curly brackets to bracket an unordered set, the square brackets to bracket an ordered set, and parentheses to indicate grouping without regard to the ordering of the parenthesized material.)

Expressing ordering on a domain will be done in one of two ways:

1. If the domain is defined as a composite data element, the ordering will be expressed as relationships between the component simple elements.
2. If the domain is defined by algorithm, that is, by a pair of algorithms, one of which generates the domain and the other of which recognizes an element in it, then ordering will be expressed by a third algorithm which tells if two elements stand in the ordering relation.

Note also that ordering a domain makes possible another means of referencing a value-string: that of giving a domain name and the ordinal position of the desired element within that domain.

Including units of measure in a domain. How do we apply our domain and representation concept to the expression of numerical units? How, for example, shall we prepare to deal with length in feet? There are two obvious choices:

1. Let the domain-name be "length-in-feet" and let the representations associated with it be numbers, say, integers;
2. Let the domain-name be "length", and let the representations associated with it be numbers (say, integers) followed by a unit name, say, inches.

From the standpoint of domain definition, both approaches are equally acceptable. The second approach, however, imposes the additional burden on the processor of recognizing that a number followed by a unit designation is to be considered a single representation.

Similarly, one might wish to have a domain of U.S. dollars. We have the choice of including the dollar sign and decimal point as part of the value-string, or we can supply it by an editing transformation when needed. If the non-numeric characters are included in the value-strings, however, the functions defined on these value-strings are somewhat more complicated to write.

Multiple value-strings for the same concept. We may wish to have multiple representation for the same concept. For example, the numbers 10 and 8 in base 8 and base 10 notations, respectively, represent the same integer. In order to be able, for output, to choose between these representations or to check them upon input we should be able to assign them to different domains. However, there is no reason that we cannot have two different but equivalent representations in the same domain. This may give rise to some ambiguities, but as far as I can tell, they will be harmless ones. (There may arise a need for some convention as to which of the equivalent representations is the "principal" one; this might be needed for debugging or output purposes where the specific domain was not explicitly specified.)

In order to be able to use equivalent value-strings interchangeably, however, we need some means of expressing their equivalence. That is, we need some explicit way of declaring that two value-strings, whether they be in the same domain or in different domains, represent the same concept. Whether two equivalent representations can be used interchangeably depends on how functions are defined on these representations; this will be discussed in more detail later under "Definitions of Functions".

Defining pairwise equivalence for sets of representations can be done either by explicit enumeration or by algorithm. For large sets, however, such as for the range of integers handled by a given processor, it is obviously unfeasible to do it by enumeration.

Special and universal concepts. There are a number of concepts with broad applicability which could be considered automatically to be members of every domain. For the purposes of permanent preservation and ready

availability for the construction of new domains, we might have these "universal" concepts stored in a domain called "UNIVERSAL". These universal concepts are discussed briefly below.

The concepts of "arbitrary" and "random" are useful for selecting an element from a composite such as a set or a list. We take the view that they are functions which can be applied to some element. In particular, they can be applied to domain designators (which name a set of value-strings) to select one of those strings. It can also be applied to the set of domain designators, in which case it selects a domain designator.

The concept of "undefined" has the nature of being an explicitly-defined member of every domain, including that of domain designators. It seems impractical to have to include the value-string of "undefined" in every domain. An alternative is to construct a special domain, which we might call the "universal" domain, in which we will put value-strings common to all other domains. Our first member of the universal domain is then the value-string "undefined". We might also have a special character reserved for this role, which we would then declare to be equivalent to "undefined".

Note that there are two kinds of "undefined":

1. domain unspecified and not uniquely determinable from value-string.
2. value-string unspecified, although domain is specified.

Note also that these two types of "undefined" can apply both to arguments and to functions. Diagnostic messages should distinguish between the 2 types of "undefined" and should indicate whether it arises from evaluation of arguments or of a function.

Other candidates for the "universal" domain include:

1. "missing", meaning"relevant but unknown"; perhaps "undefined" is good enough for this purpose;
2. "non-existent entity": empty (cardinality zero), no correspondent;
3. "structure undefined";
4. "structure improper";
5. "inconsistent", contradictory; overdefined;
6. "ambiguous", as might result from asking for "the name" of an element which has more than one name;
7. "unrestricted" or "any"—this will have applicability as a value of a restrictor;

8. "unspecified" means "derivable from other information" as is the domain designator associated with a value-string which exists in only one domain.
9. "null", an element of cardinality 1, as distinguished from "empty", above, which has a cardinality of zero. The concept of null is extremely useful for constructing recursive definitions.

SIMPLE DATA ELEMENTS

"Cells", "variables", and "constants". A "cell" we define to be a storage slot of unspecified size which contains exactly one value-string. (If no value-string has been put into it by the user, then it must contain one which is interpreted by the processor as meaning "undefined".) Words sometimes used to mean what we define as "cell" are "variable" and "atomic element". A cell may have identifiers associated with it, by means to be discussed later.

There is a distinction between a "variable X" and a "constant X". If X is a variable, "X" is the identifier of a slot containing some value-string which is called the "value of X". The "constant X", however, is a value-string. It does not have an associated identifier and it cannot be modified in the sense that a variable can be modified.

When a variable is assigned a value, that is, when a cell has a value-string put into it, we say the variable is "bound". It is then no longer variable, it is constant (although temporarily so, perhaps). A variable, or cell, is not a constant in the sense of being a value-string. Rather it contains a constant, or value-string.

Simple data elements. A simple data element, as distinguished from a composite one (to be defined later), is composed of one or more cells. A simple data element can be thought of as one which expresses a single value of a variable, a single concept. It corresponds roughly to the idea of "simple variable" in ALGOL.

One of the cells of a simple data element holds the "principal value-string" of that data element. Other cells, tied to the principal value cell in specified relationships, can hold such entities as identifiers, domain designators, and other pieces of information "about" the principal value-string. This definition of "simple data element" may be confusing because the cells tied to the principal value cell can themselves be simple data elements. In general these latter are incomplete simple data elements.

If they were not, a simple data element would have a set of simple data elements giving information about each, and so on, ad infinitum.

A cell can have associated with it, in various naming relationships, a set of strings from the domain of "identifiers". Examples of such naming relationships are: ordinary-name-of, principal-name-of, class-name-of.

A domain designator can stand in either of two relationships to a cell: it can be in the relationship of "domain descriptor", or "domain prescriptor". A domain descriptor is used when the domain of the value-string cannot be discovered by inspection: that is, when the value-string occupying the cell is not unique to a single domain. A domain prescriptor is used to prevent the cell from being assigned (from containing) a value-string from an unwanted domain; ultimately this helps to prevent the computation of "garbage". In other words, the value-strings which stand in a domain prescriptor relationship to a given cell specify the domains from which value-strings can be accepted. A domain prescriptor is not limited to being a single domain designator, however; it can be a set of domain designators, which are then interpreted as being alternative possibilities.

Some intuitive (informal) illustrations of these concepts are given in Figure 4-1.

Constants. A constant is a value-string, stored as the constants of some cell, and having an associated identifier. A constant may be protected from "damage" (inadvertent alteration) by having some protection indicator attached and by having the referencing mechanism make a check for this indicator. This indicator is under the user's control, so that he can turn protection on and off as he wishes.

Identifiers. There is a very special class of value-strings which serve as names of data elements. Henceforth we call these names "identifiers". The reason that identifiers are singled out for special mention and treatment is that as a class they provide the fundamental technique by which reference is made to data elements. However, individual identifiers are not the only device for being able to reference (access) a data element. The whole issue of referencing will be discussed in detail in Chapter 5.

An identifier is some string of characters of the alphabet. To avoid parsing problems, I arbitrarily restrict identifiers to be strings not

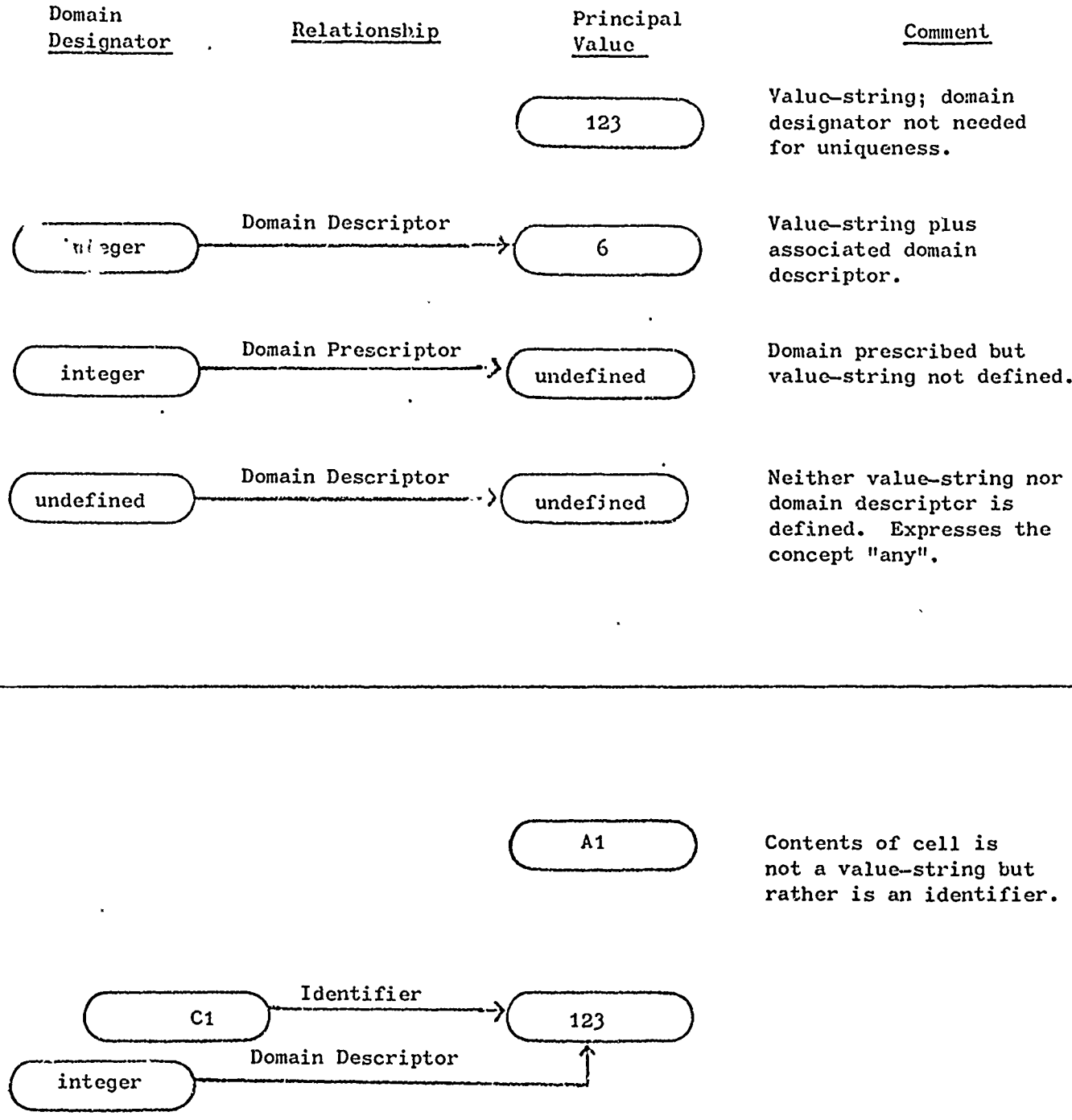


Figure 4-1. Examples of Simple Data Elements

containing blanks. The spelling of an identifier can be changed by programmable action, though not without running a risk of adverse consequences.

I take the point of view that the identifier of a data element is another element that stands in an "identifier relationship" to the first element. I say, for example: "The string"C1" stands in the identifier relationship to the number represented by the string "123". The identifier relationship is not fixed; that is, a given string considered to be an identifier can become "un-related" (in the sense of dissociated) to one data element and be put into an identifier relationship with another data element.

The relationships of naming by identifiers I take to be primitive, because there is no way in general of referring to a data element in order to explicitly state that it stands in an identifier relationship. We will be able to attach any string (having no imbedded blanks) as an identifier of a storage cell. How this attachment is to be represented in a processor is undefined in the language. More precisely, it is undefined in the user language, though of course it must be defined in some way so that the processor behaves appropriately when it receives one of the following directions:

1. Access the cell whose identifier is the (named or exhibited) string_____;
2. Exhibit the string which identifies the data element which has the following properties_____.

A string can be made to stand in an identifier relationship to more than one cell, as pictured in Figure 4-2. This makes the identifier an ambiguous one. Used by itself it involves a set of elements (but it does not name an element which is a set). It is up to the user to be aware of any ambiguities he may create, and provide for their resolution when necessary. To access a cell which has an ambiguous identifier, the ambiguity must be resolved by some information in addition to the ambiguous identifier. For example, a pair of ambiguous identifiers may have a unique intersection: that is, there may be only one cell to which they both stand in the identifier relationship.

Several identifiers can stand in the identifier relationship to a single element, as pictured in Figure 4-3. We say that these identifiers are "synonyms" with respect to that element. One of these identifiers can be made to stand in the relationship of "principal-identifier" to an

element. This latter provision is to make possible the obtaining of a consistent answer to the question "what is the principal identifier of the element having the following characterstics_____?" Notice that we may have a complex situation, pictured in Figure 4-4, in which "A" is the principal identifier of element 1 and an ordinary identifier of element 2, while "B" is the principal identifier of element 2 and an ordinary identifier of element 1.

If strings "A" and "B" both stand in the same identifier relationship to the same data element, "A" and "B" are synonyms. That they stand in the synonym relationship to each other is an implied relationship, but it is not explicit unless we make it so. The language processor is not going to make logical deductions for us!

The set of all possible identifiers forms the domain of identifiers. This domain will necessarily be defined by an algorithm because it is too large to enumerate. The set of identifiers actually used in a given program, however, must be explicitly enumerated and stored in some special place known to the processor, for this set of identifiers must be examined each time a reference is made to a data element by giving its identifier. How a given identifier is "tied", or "related", to the data element it identifies will be explained later.

Since each identifier used is actually stored, it has the status of a data element. As a data element, it is accessible to the user; he can inspect it, modify it, or replace it. One might think that in order to access a data element which is an identifier one would need to reference it by yet another identifier; this is possible but not necessary. An obvious way to reference an identifier is to exhibit the identifier strings in quotes, since quoting a string makes it into a name for itself. There are yet other ways to reference identifiers, which are based on relationships to other known data elements; how this can happen will become clearer later.

Since identifiers are data elements, they can be created, accessed, modified, associated with and dissociated from other data elements. Such abilities are in distinct contrast to those in current programming languages which are designed to be compiled. In such languages, identifiers expressed in the source language are "lost": they are converted to machine addresses by the compiling process and are thus made inaccessible to manipulation by a program.

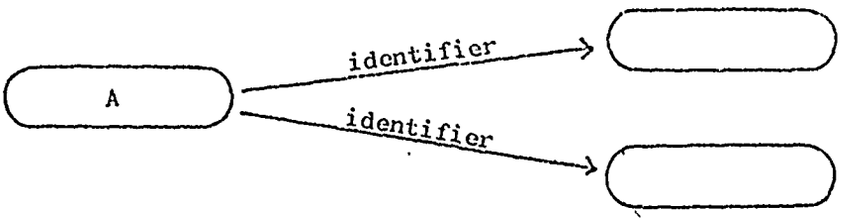


Fig. 4-2 Ambiguous Identifier

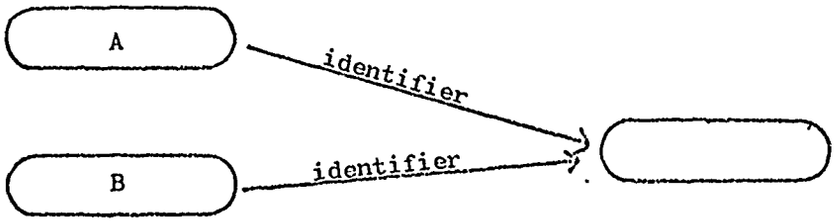


Fig. 4-3 Multiple Identifiers

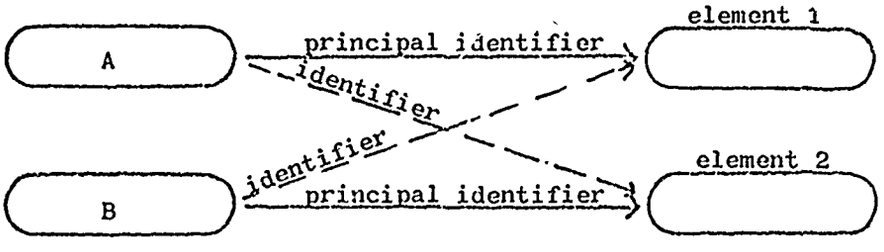


Fig. 4-4 Complex Situation Involving Multiple Identifiers

There will be occasions when a data element will need an identifier. In such a case, if an identifier is not supplied by the user, it must be supplied automatically by the processor. On the occasions when an identifier is generated by the processor, the generation must be done in such a way as to avoid conflict with existing identifiers. This conflict could be avoided by the processor's using one or more characters reserved for the purpose. This is undesirable, however, because the user should be able to ask: "What is the identifier of the element designated _____?" or "What is the identifier of the element having the characteristics_____?"

On occasion the programmer could accidentally generate or specify an identifier that the processor had generated automatically. If we wish to avoid this, we must either tell the programmer how to avoid accidentally picking an identifier that could be generated, or we must make every introduction of an identifier give rise to a check of all existing identifiers. The former is undesirable, and the latter, in the general case, is costly to implement. A compromise, but not a foolproof system, is to have the generated identifiers be of such composition that no human being would normally conceive of it as an identifier: for example, such unlikely sequences as "XXX.1G".

Unless prohibited by formation rules for identifiers, a given string may both be an identifier of a cell and be a value-string. Which interpretation it should have in a given instance must be determined from context. For example, if a string is an argument of a function cell, the definition of the function may determine whether that argument is to be interpreted as an identifier or a value-string. It is less confusing, however, if a given string is not used both as an identifier and as a value-string.

The concept of "indirect name" is readily seen to be represented by an element standing in an identifier relationship to a second element which in turn stands in an identifier relationship to a third element. The first element can then be said to be an indirect name of the third element. Such a chain of elements linked by an identifier relationship could be arbitrarily long. In such a multiple-link chain, the number of steps to be taken from an element, regarded as an identifier, to its correspondent could be given in any of several ways:

1. One might trace identifier relationship steps as far as they go.

2. One might expect to give the number of identifier relationship steps to go.
3. One might find an explicit "indirection indicator" attached to some elements.

I have not said the last word on identifiers. Later I will deal with two more issues:

1. The association of an identifier with a composite data element.
2. The association of an identifier, or marker, with a point "between" two succesive data elements.

Review. The following example shows in more detail how we can use the concepts explained thus far. We assume the existence of a processor which has the defining capabilities that are needed, but which has as yet no particular domains or functions defined other than the primitive domain of a given alphabet. Assume that the following statements are steps in a program:

1. Choose strings "T" and "F" as value-strings of a new domain.
2. Associate the identifier "Boolean" as a name of this new domain.
3. Define a function on this Boolean domain as follows:
   a. arguments 1 and 2 called by value (in the ALGOL sense)
   b. result is from the domain "Boolean"
   c. function table is:

arg1	arg2	result
T	T	T
T	F	F
F	T	F
F	F	F

4. Associate the identifier "AND" as the name of this new function.
5. Evaluate the function AND ("T", "T"); answer "T".
6. Evaluate the function AND ("F", "TRUE"); answer: "undefined", because the argument "TRUE" is not yet defined.
7. Add to this "Boolean" domain the value-string "TRUE".
8. Let "T" and "TRUE" be equivalent.
9. Evaluate the function call AND ("TRUE", "T"); answer "T".
10. Add to the "Boolean" domain the value-string "U".
11. Evaluate the function call AND ("U", "F") answer: "undefined", because the function hasn't been defined for this combination of arguments.

12. Add to the function table for the function "AND":

arg1	arg2	result
T	U	U
F	U	F
U	T	U
U	F	F
U	U	U

13. Evaluate the function call AND ("U", "F"); answer "F".

In this example we defined the concept of Boolean variable, then defined the Boolean function AND, and then extended the domain and the definition of the function to include a new member of the domain. Try that in your favorite programming language!

The reader has been introduced to the idea of setting up arbitrary domains of value-strings and defining functions upon them. These are the most trivial, fundamental concepts, yet few, if any, programming languages provide for them. Current programming languages provide a few, fixed domains: letters, digits, and possibly a few other characters and strings of these; decimal integers up to some maximum arbitrarily chosen to match the word length of some hardware machine; a set of rational numbers corresponding to that set of quantities manipulatable by some "floating-point" circuitry of the same hardware machine; a 2-valued, or "Boolean" domain. It should be painfully obvious that most current languages lack the facilities for defining new domains, for representing the concept of "undefined", for constructing identifiers freely.

How, in current languages, do we define, or simulate, a new domain? We may be lucky enough to find that the domain we want is a subset of an already-defined domain. If we aren't so lucky, we will have to map mentally the domain we want into a domain that we have. And this seemingly simple mapping can be very troublesome. Consider what we must do if we want a domain of "true," "false," and "undefined". We might map these into a restricted integer domain of 0, 1, and 2, or perhaps -1, 0. and +1. Then we would have to redefine all the conventional Boolean operators ("and", "or", and "not", etc.) as operators on a set of numbers. This is possible, and workable, but an unnatural and confusing situation.

COMPOSITE DATA ELEMENTS. ILLUSTRATED

Introduction to composites. For all but the most trivial computations, data elements are needed which are not individual data elements but which

are sets of elements associated in some way. Such a set of related elements we shall call a "composite data element". A simple example of a composite data element is a complex number, which is an ordered pair of two numbers, the components of the composite being termed the "real part" and the "imaginary part", respectively. A second example of a composite is a human family. Such a composite has, usually, three components: termed a "father", a "mother", and "children", where children in turn is another composite, a set each of whose members is a "child".

The design and storage arrangement of composites is a non-trivial undertaking. In fact, for some complex data processing problems, it is probably the activity that takes the majority of the analyst's effort. This section intends to explain what is needed in the way of methods of dealing with composites and to show one way of attaining the desired end. Because the subject of composites can be complex and difficult, I am initially going to avoid definitions and abstractions, and present instead a series of examples. Thus I will exemplify the requirements before formalizing them.

When I am describing composite data elements I must indicate which of two kinds of description I am using: one kind describes the individuals in the class; the other kind describes a single composite as if it were the only instance of a class. The first kind of description I will say concerns a "model" and its instances, and I will refer to it as the "model" type of description. An example of the model type of description is the record description as used in COBOL and PL/I, which gives a record layout. Each record of the file to which the record description belongs is an instance of this record description or "model". The second kind of description is that of a lone instance which is not a member of a class of similarly-structured composites, and I will refer to it as a "modelless-instance" type of description. An example of this modelless-instance type of description is the occurrence of a list structure in IPL-V. It applies to exactly one structure, not to a class of similar structures. Current programming languages usually provide for only one or the other of these types of description, and rarely, if ever, for both.

As the first example I choose a calendar date with a specific meaning, say, "calendar date of latest revision". This will be given in the modelless-instance type of description. (Later examples will be presented

which are constructed around the concept of the class of calendar dates.)

A series of examples based on this composite are given in order to illustrate a variety of points. In all these examples I shall observe the following conventions:

1. Expressions in capitals represent expressions which can be considered as actually stored.
   a. Expressions without quotes are considered to be names: that is, they could be found in some list of identifiers.
   b. Expressions in quotes are "value-strings".
2. Expressions enclosed in square brackets are explanatory material which is not explicitly stored.
3. Where there is no notation to the contrary, sequence is understood to be material. Where sequence does not matter, either the notation "set" appears, or the conventional curly brackets { and } will be used.

Examples. The series of examples labeled 1a, 1b, etc., are concerned with the modelless-instance type of composite in which the parts are not referenceable except by ordinal position. (That is, the components are not explicitly named.)

The second series of examples, 2a, 2b, etc., introduces identifiers of components and illustrates the fact that when the components have identifiers, the sequence of the components may not matter.

The third series of examples, 3a, 3b, etc., introduces models which apply to a fixed sequence of components in the instances.

The fourth series of examples, 4a, 4b, etc., introduces models which describe their associated instances but in which the components do not necessarily have a 1-to-1 correspondence with the components of the instances.

The fifth series of examples, 5a, 5b, etc., illustrates complex("multiple-level") composites, and the fact that the concepts of model and modelless instance can be used together.

The sixth series of examples, 6a, 6b, etc., shows how relationships and relations can be handled as composites.

The seventh series, example 7a, discusses the concept of models of models.

Example 1a.

[Identifier]	REV-DATE
	[Value-String]
[First component: year]	"1967"
[Second component: month]	"APRIL"
[Third component: day]	"15"

Thus we have a three-component composite. The components are unnamed (that is, their names are not explicitly stored). The components are accessible (referenceable) only by giving their ordinal positions in the composite. Thus, the month component must be referred to as "component 2 of REV-DATE".

Example 1b.

[Identifier]	REV-DATE
[Identifier]	DATE.OF.REVISION
	[Value-String]
[First component: year]	"1967"
[Second component: month]	"APRIL"
[Third component: day]	"15"

Here we have two synonymous identifiers for the same composite as illustrated in example 1a above. The "synonymity relation" exists only with the respect to this particular composite. It could be the case that DATE.OF.REVISION, but not REV-DATE, is an identifier of some component in some other composite.

Example 1c.

[Identifier]			REV-DATE	
[First component: year]	[restrictor]	"INTEGER"	[Value-String]	"1967"
[Second component: month]	[restrictor]	"MONTH"	[Value-String]	"APRIL"
[Third component: day]	[restrictor]	"INTEGER"	[Value-String]	"15"

Here we have added a new dimension of information: a restriction as to the nature of the value-strings which are legal components. These restrictors specify that the value of the first and third components must be from the domain of integers, and that the value of the second component must be from the domain of months. (We assume that these domains have already been defined.)

Example 1d.

[Identifier]			REV-DATE	
[First component: year]	[restrictor]	"INTEGER"	[Value-String]	"1967"
[Second component: month]	[restrictor]	"MONTH" or "INTEGER"	[Value-String]	"APRIL"
[Third component: day]	[restrictor]	"INTEGER"	[Value-String]	"15"

Here we have introduced another concept; the restriction of a value-string not to a single domain but to a set of alternative domains. In the example I have restricted the second component to be expressed either as a month name or as an integer (that is, month number).

Example 1e.

[Identifier]	REV-DATE [Value-String]
[First component: year]	"1967"
[Second component: month]	"APRIL"
[Third component: day]	"15"
[Fourth component: change marker]	"RESET"

This is one way to have a marker, or indicator, associated with the date, which marker can be programmed to indicate if the date has been changed in some prescribed time interval.

Actually, this technique is intellectually unsatisfying. This technique implies that the marker is part of the date, which is nonsense. The marker is "associated" with the date in a relationship which we could call "'a' is a change-indicator of 'b'". We would like some appropriate way of explicitly showing this relationship, so that we could get programmed answers to such queries as:

1. "Does anything stand in the relationship of 'change-indicator' to 'REV-DATE'?"
2. "What is the 'value' of the element standing in the relationship of 'change-indicator' to 'REV-DATE'?"

Example 1f. It might happen that we want our calendar date to be available in two different sequences depending on its use. The year-month-day sequence is preferable for computational purposes, where we might be incrementing the date by a given time interval between this date and another one. For printout purposes, however, a different sequence may be preferable, such as the government standard sequence: day-month-year.

The important concept here is that a single set of values should be viewable as two separate composites. The concept of a composite thus cannot be a single set of storage cells. It must instead be a referencing scheme to an arbitrary set of storage cells which does not preclude applying other referencing schemes to those same cells.

The following illustrates in a very informal way the concept of having two views of the same composite:

[Identifier]	REV-DATE		[Identifier]	REVISED
	[Value-String]			[Value-String]
[First component]	[year]	"1967"	[day]	"15"
[Second component]	[month]	"APRIL"	[month]	"APRIL"
[Third component]	[day]	"15"	[year]	"1967"

Example 1g. A situation could arise in which the "day" component of the date was not given. Assuming that the composite structure for REV-DATE was already set up, temporarily omitting the third component, and possibly having to restore it later, is a bit clumsy. In such a case we need a value-string such as "undefined" which represents the fact that no value-string representing a day has been supplied.

[Identifier]	REV-DATE
	[Value-String]
[First component: year]	"1967"
[Second component: month]	"APRIL"
[Third component: day]	"UNDEFINED"

Example 1h. In the preceding examples, the components of the composite have been simple data elements. The components need not be limited to simple elements, however; they can be composite. If an arbitrarily complex composite, A, is to be a component of a higher-level composite, B, the reasonable way to represent A within B is by its identifier.

To illustrate, suppose a composite representing a transaction, involving a sender, receiver, and transaction-date. Sender and receiver can be thought of as simple elements, while transaction-date can be regarded as a composite, identical in structure to the revision-date shown in example 1a. We could represent this with two separate composites as follows:

[Identifier]	TRANS-DATE	[Identifier]	TRANSACTION.1
	[Value-String]		[Value-String]
[year]	"1967"	[sender]	"JOE SMITH"
[month]	"JUNE"	[receiver]	"TOM JONES"
[day]	"21"	[date]	TRANS-DATE

Example 2a.

[Identifier] REV-DATE

	[Part-Identifier]	[Value-String]
[First component]	YEAR	"1967"
[Second component]	MONTH	"APRIL"
[Third component]	DAY	"15"

Here the components have individual identifiers so that a component can be accessed by giving its component identifier as well as the main identifier. For example, to access the value-string "APRIL", one could refer to it as "month of REV-DATE".

Example 2b. With the part-identifiers explicit as they are in example 2a, there is no need for a fixed sequence of components. The composite can be represented without any implied sequence of components thus:

[Identifier] REV-DATE

[Part-Identifier	Value-String]
YEAR	"1967"
DAY	"15"
MONTH	"APRIL"

This form corresponds to that of the "property list" of IPL-V, the "association list" of LISP, and the "description list" of FORMULA ALGOL. The contents of such a list need not be regarded as properties or descriptions. We will treat property lists later in this series of examples.

Example 2c.

[Identifier] REV-DATE

	[Part-Identifier]	[Value-String]
[component]	{YEAR, YR}	"1967"
[component]	{MONTH, MO}	"APRIL"
[component]	{DAY, DA}	"15"

Here the components have multiple individual identifiers. Now a component can be accessed by giving any one of its components or part identifiers in addition to the main identifier. For example, to access the

value-string "APRIL" one could refer to it equally well either as "MONTH of REV-DATE" or as "MO of REV-DATE".

Any of the part-identifiers may be omitted, leaving gaps in the above layout. If the layout is constrained to be regular, the gaps must be filled with some string such as "UNSPECIFIED".

Example 2d.

PART-IDENTIFIER: YEAR RESTRICTOR: "INTEGER" VALUE-STRING: "1967"
PART-IDENTIFIER: MONTH RESTRICTOR: "MONTH" VALUE-STRING: "APRIL"
PART-IDENTIFIER: DAY RESTRICTOR: "INTEGER" VALUE-STRING: "15"

Here have been made explicit the types of association, or "roles". Now we are completely freed of the necessity for regularity. We could omit any of the above pairs without introducing confusion.

Example 2d1. Because the component description of example 2d is quite regular, we might be tempted to rewrite it as follows:

[Identifier] REV-DATE	[RESTRICTOR]	[VALUE-STRING]
[First component: year]	"INTEGER"	"1967"
[Second component: month]	"MONTH"	"APRIL"
[Third component: day]	"INTEGER"	"15"

We have taken advantage of the regularity and have in some sense effected an economy of description. However, with this economy we lose the freedom to omit any part. We must substitute the string "unspecified" or "undefined" for any missing part.

Example 2d2. But suppose we had a composite like 2d1, with some irregularity:

[Identifier] REV-DATE			
[First component: year]	[RESTRICTOR:] "INTEGER"	[VALUE-STRING:]	"1967"
[Second component: month]		[VALUE-STRING:]	"APRIL"
[Third component: day]		[VALUE-STRING:]	"15"

We can easily force it to be regular; in the style of example 2d1:

[Identifier] REV-DATE	[RESTRICTOR]	[VALUE-STRING]
[First component: year]	"INTEGER"	"1967"
[Second component: month]	"UNSPECIFIED"	"APRIL"
[Third component: day]	"UNSPECIFIED"	"15"

Fu. in doing so we have been forced to specify "unspecified" which is a nuisance, and in some sense must waste storage.

Example 2e.

[modelless instance]

[Identifier] A A MFG.

[Part-Identifier]	[Value-String]
NAME	"A A MFG"
STREET	"1400 MARKET STREET"
CITY	"WILMINGTON"
STATE	"DELAWARE"

This example shows that a given string, here "A A MFG", can be used both as an identifier and as a value-string. As long as we know from form or from context which of these two interpretations to give a string, we do not have ambiguity. We can even reference the "NAME of A A MFG", which is, naturally enough, "A A MFG".

Example 3a.

[Model]		[Instance 1]		[Instance 2]	
[Identifier]	DATE	[Identifier]	REV-DATE	[Identifier]	ORIG-DATE
		[Model name]	DATE	[Model name]	DATE
		[Value-String]		[Value-String]	
First component identifier:]	YEAR	"1967"		"1967"	
Second component identifier:]	MONTH	"SEPTEMBER"		"MARCH"	
Third component identifier:]	DAY	"23"		"2"	

Here we have a "fixed" model, in the sense that the number and sequence of the components are fixed. A model is thus a rule concerning the component parts of a set of like composites ("instances"). At a minimum, the model provides for identifiers of the parts. It can also provide for any other information which applies uniformly to the components of instances individually.

Notice that a model itself is a composite data element. This suggests that a set of similar models may themselves have a model. Later we shall see that such is indeed possible.

Example 3b.

[Model]
[Identifier] DATE

[Instance 1]
[Identifier] REV-DATE
[Model Name] DATE

[Part-Identifier]		[Restrictor]	[Value-String]
[First component]	YEAR	"INTEGER"	"1967"
[Second component]	MONTH	"MONTH"	"SEPTEMBER"
[Third component]	DAY	"INTEGER"	"23"

This model possesses domain-restrictors for each component. Other types of information that apply individually and uniformly to all the corresponding components of all instances are: initial values, access-controls, range limits on value-strings in an ordered domain. Information which applies collectively must be handled in a different way (see example 6a). Information which does not have the same value for all of the nth components must also be handled differently (see example 3c).

Example 3c. Suppose we wished to have a "changed-marker" associated with each component of each instance. The contents of such markers cannot be stored in the model, since each marker is associated independently with each component of an instance. That is, the contents (value-string) is not necessarily the same for the corresponding component in all instances. We need the following arrangement:

[Model]
[Identifier] DATE

[Instance 1]
[Identifier] REV-DATE
[Model Name] DATE

		[Changed-marker]	[Value-String]
[1st component identifier]	YEAR	"UNCHANGED"	"1967"
[2nd component identifier]	MONTH	"CHANGED"	"SEPTEMBER"
[3rd component identifier]	DAY	"CHANGED"	"23"

Example 3d. Certain kinds of models, called "skeletons", have the interesting property that every instance of the model starts out as a copy of the skeleton. That is, before any of its value-strings are modified, an instance is simply a copy of the skeleton. The kinds of information that would go in a skeleton are structural information and values which are initially the same for all newly-created instances but which thereafter are subject to change. Examples of these are: initial value-strings, changed-markers, and slot allocation. Since part-identifiers

3rd component of TRACT-4. However, the members of a set are understood to have no fixed sequence within that set, hence a later access to the 3rd component of TRACT-4 will not necessarily access the same element as before.

Example 3f.

[Model]		[Instance 1]		[Instance 2]	
[Identifier]	BOOK	[Identifier]	B1	[Identifier]	B2
		[Model name]	BOOK	[Model name]	BOOK
[Part-Identifier]		[Value-String]		[Value-String]	
TITLE		"SYNTAX"		"ROOT TABLES"	
AUTHOR		"SMITH, J.P."		"JONES, X,X."	
PUBLISHER		WILEY		WILEY	

[Model]		[Instance]	
[Identifier]	PUBLISHER	[Identifier]	WILEY
		[Model name]	PUBLISHER
[Part-Identifier]		[Value-String]	
NAME		"JOHN WILEY & SONS, INC."	
CITY		"NEW YORK, N.Y."	

This illustrates the occurrence of a composite WILEY as a component of two other composites (B1 and B2). This occurrence of a component common to two or more composites is what takes the data representation scheme out of the pure hierarchy, or "tree," category.

Example 4a. It is not always appropriate to use a model having a fixed structure. It may be desirable to have instances which contain components chosen from a specified overall set of components. In such a case the model may specify the overall set. Clearly it is necessary for each instance to identify explicitly which components are present. In this example, this identification has been accomplished by giving explicit part-identification.

[Model]		[Instance 1]	
[Identifier]	DATE	[Identifier]	REV-DATE
		[Model name]	DATE
[Part-Identifier]	[Repetition factor]	[Part-Identifier]	[Value-String]
YEAR	"1"	YEAR	"1967"
MONTH	"0 or 1"		
DAY	"0 or 1"		

and domain-restrictors are not subject to change on a per-instance basis they are not appropriate to a skeleton. Both a model and a skeleton may be used with a given set of instances. Here is an illustration:

Model		Skeleton		Instance	
[Identifier]	DATE	[Identifier]	DATE-A	[Identifier]	REV-DATE
[Skeleton name]	DATE-A	[Model name]	DATE	[Model name]	DATE
		[Value-String]		[Value-String]	
[1st component identifier]	YEAR	"1967"		"1967"	
[2nd component identifier]	MONTH	"UNDEFINED"		"UNDEFINED"	
[3rd component identifier]	DAY	"UNDEFINED"		"UNDEFINED"	

If the part-identifiers were not needed, the model in this example could have been omitted.

Example 3e. How do we give a name to a set of elements? In some cases this set is the set of instances associated with a model. But there will be cases in which the members of the desired set either:

1. are not all the instances of a given model;
2. belong to one of several models;
3. are modelless instances;
4. are some combination of 1,2,3 above.

To sum up, we need a method of specifying arbitrarily-chosen elements to be members of a set. One way to do this is to consider that we have a primitive model called a "set". "Primitive" means that the concept of "set" is not defined in the user language. Any set we wish is then an instance of this model, and is essentially a list of identifiers of members of the set. An example of a set is:

[Identifier]	TRACT-4
[Model name]	SET
[Value-String]	
	HOUSE-A
	HOUSE-B
	HOUSE-C
	HOUSE-D

House-A, House-B, etc., are the identifiers of composite data elements, each of which might be an instance of a model called HOUSE. Since at any given time there exists some sequence of the components of a set, the components of their identifiers can be accessed by ordinal position, e.g.;

[Instance 2]

[Identifier] PUB-DATE
[Model name] DATE

[Part-identifier]	[Value-String]
MONTH	"APRIL"
YEAR	"1967"

[Instance 3]

[Identifier] ORIG-DATE
[Model name] DATE

[Part-Identifier]	[Value-String]
MONTH	"APRIL"
DAY	"15"
YEAR	"1967"

The model illustrated here says, in effect:

1. The component YEAR must occur once.
2. The components MONTH and DAY are optional.

Note that the sequence of the components is immaterial, because each component explicitly contains its associated part-identifier.

Example 4b. We may want a structure which is a set of similar components but for which the number of components is not known until after program execution has begun. An example might be a specific set of children, illustrated thus:

[Identifier] CHILD-SET	[Identifier] CHILD-SET-1	[Identifier] CHILD-SET-2
	[Model name] CHILD-SET	[Model name] CHILD-SET
[Part-Identifier] CHILD	[Value-String]	[Value-String]
[Repetition factor] any	"JOE"	"MARY"
		"FRANK"
		"BETTY"

The data-definition language of Standish calls this "indefinite replication".

Example 5a.

[Identifier] SALE-1

[Part-Identifier]	[Value-String]
SELLER	"JOE"
BUYER	"MAX"
DATE	DATE-1

[Model]

[Identifier]	DATE
[First component]	YEAR
[Second component]	MONTH
[Third component]	DAY

[Instance 1]
[Identifier] DATE-1
[Model name] DATE
"1967"
"APRIL"
"15"

This example illustrates two things:

1. Composites can have as components other composites.
2. A component which is composite need not be of the same kind as its overall or parent composite. Here the parent composite is a modelless instance while one of its components is a modelled instance.

It is also true that the components of a given composite need not be all of the same kind: there can be a mixture of modelled and modelless instances.

Example 5b. There is a fine but important distinction between two uses of a component which has several members. Consider this example:

[Identifier] ITEM-1	
[Part-Identifier]	[Value-String]
TITLE:	"BUGS"
AUTHOR:	"J.J. JONES"
PUB-DATE:	"1963"
DOC-NO:	D1

[Identifier] D1 [Model name] ALTERNATIVE-SET	
[Value-String]	
"TR-103"	
"AD 000 122"	
"SP-1066"	

In order to be able to locate this composite by searching, we want to be able to say something like "that composite which has a (presumably unique) value-string of 'SP-1066' as a value of DOC-NO". Further assume we do not know in advance whether DOC-NO has as its value a single value-string or a set. We could test each occurrence of DOC-NO to find out whether its correspondent was a single value-string or was a set. This is rather clumsy. What we want is the flexibility of having a component (DOC-NO) for document number which can have any number of associated value-strings without having to make an artificial distinction between the cases of 1 and more-than-1. It should be noted that there is an alternative, and equivalent method of expressing groups of similar components, as exemplified below:

[Identifier]	ITEM-1
[Part-Identifier]	Value-String
TITLE:	"BUGS"
AUTHOR:	"J.J. JONES"
PUB-DATE:	"TR-103"
DOC-NO:	"AD 000 122"
DOC-NO:	"SP-1066"
DOC-NO:	

Example 5c. Certain information applies to a set of instances collectively rather than individually. Examples of such information: The number of instances, the "highest value" of a value-string of a specified

component of all the instances, and arbitrary properties assigned to the set of instances. Such information goes by the name of "property list" (IPL-V), "association list" (LISP), or "description list" (FORMULA ALGOL). We call it "summary information". There are two fundamentally different ways of attaching summary information to a set, which here we will call the "data set", to distinguish it.

The first method is to consider the summary information as a composite, the data set as a composite, and the first composite standing to the second composite in the relation "property set of".

[Identifier] PROPERTY-RELATIONSHIP-1
[Model name] SET

PROPERTIES-A
TRACT-1

[Identifier]	TRACT-1	[Identifier]	PROPERTIES-A
[Model name]	SET	[Model name]	SET
		[Part-Identifier]	[Value-String]
	HOUSE-A	LOWER-PRICE:	"$18,200"
	HOUSE-B	UPPER-PRICE:	"$22,500"
	HOUSE-C	SCHOOL-NAME:	"HUGHES"
	HOUSE-D		

This approach says, in effect, the composite named PROPERTIES-A is a list of properties which apply collectively to the set (of houses) named TRACT-1. The second approach is to make a composite of the properties, as individual components, along with the name of the data set.

[Identifier]	TRACT-DATA-1	[Identifier]	TRACT-1
[Model name]	SET	[Model name]	SET
[Part-Identifier]	[Value-String]		
Lower price:	"$18,200"		HOUSE-A
Upper price:	"$22,500"		HOUSE-B
School name:	"HUGHES"		HOUSE-C
Tract-Ident:	TRACT-1		HOUSE-D

Example 5c1. A possible variation on example 5c is to replace the component pair

LOWER-PRICE: "$18,200"
UPPER-PRICE: "$22,500"

with PRICE-RANGE: RANGE-1

where RANGE-1 names a separate composite as follows:

[Identifier] RANGE-1
[Model name] SET

[Part-Identifier]	[Value-String]
LOWER-PRICE:	"$18,200"
UPPER-PRICE:	"$22,500"

Example 6a. Suppose we wanted to express a relationship among several entities, where by "relationship" we mean an instance of a relation, and by "relation" we mean a set of relationships. Let us name these entities A,B, and C without saying what they actually consist of. (They might be names of cities, for example). Then an instance would be:

[Identifier] X1
[Model name] LIST

[Value-String]
A
B
C

This could be a relationship which is an instance of the relation "between". This relationship corresponds to the sentence "B is between A and C". The reader is reminded that the mathematical definition of a relation is a set of n-tuples (I call it the set of instances), where each n-tuple is a list (ordered set) of names of entities which stand in that relationship. Note that we could give component identifiers to the parts of a relationship:

[Identifier] X1
[Model name] SET

[Part-Identifier]	[Value-String]
LEFT-OBJ	A
MIDDLE-OBJ	B
RIGHT-OBJ	C

We might prefer a form in which the model was explicit:

[Model]
[Identifier] BETWEEN

[Part-Identifier]
LEFT-OBJ
MIDDLE-OBJ
RIGHT-OBJ

[Instance]
[Identifier] X1
[Model name] BETWEEN

[Value-String]
A
B
C

If we wanted the relationship to be symmetric we could have instead:

[Instance]

[Identifier] X1
[Model name] LIST

[Part-Identifier]	[Value-String]
END OBJ	A
END OBJ	C
MIDDLE OBJ	B

Example 6b. A slightly different viewpoint is to regard the "between" relationship as a special case of linear ordering. For an ordered set of components we have the primitive model LIST. We could express the fact that B is between A and C by writing

[Model name] LIST

A

B

C

Notice that here an identifier is not needed. We in general do not associate identifiers with individual relationships. The meaningful entity to have an identifier is the set of all the individual relationships; this identifier would most sensibly be the name of the relation concerned.

Example 7a. Since a model is itself a composite data element, one might conjecture that a model could itself have a model. This is indeed the case. It is possible to go another "level", and have a model of a model of a model. It is not clear that this latter capability is necessary, however. It appears that one can supply the second-level model with enough primitives to take care of almost any conceivable situation.

A very rough indication of this concept of a model of a model is given in the example below:

[Instance]

[Identifier] P1
[Model name] PERSON

[Part-Identifier]	[Value-String]
NAME	"JOE WILLIAMS"
SEX	"MALE"
BIRTHDATE	"4/15/27"
STREET	"12 POST ROAD"

[Model]

[Identifier] PERSON
[Model name] MODEL-2

[Part-Identifier]	[Domain Restrictor]	[Repetition Factor]
NAME	STRING	1
BIRTHDATE	SHORTDATE	0 or 1
STREET	STRING	0 or 1
SEX	SEX-ABBR	0 or 1

Note: this model has to say, in a way not shown here, that every related instance of this model is a set of "Part-Identifier: value-string" pairs.

[Model of model]

[Identifier] MODEL-2
[Model name] -

.....

Note: This model has to say, in a way not shown here, that every related instance of this model is a set of"Part-Identifier: domain-restrictor: repetition-factor" triples.

The foregoing examples of composite data elements are intended only to be illustrative. They lack completeness and rigor. For example, in example 3b, I have not indicated how the different components of the model are to be recognized. In example 3c I have not shown how the parts of the instance are to be recognized.

These examples try to convey the range of variation of data structures. They have not illustrated all possible combinations of features. To have done so would have been needlessly confusing. When all possible combinations are considered, there is a very large number of data structures. A truly flexible language must provide for them all. Such a language will be practical only if it can provide for this wide variety of structures by a simple but general technique. If it cannot, then either it will accommodate too few structures, or it will be cumbersome through having too many individual ("ad hoc") types of data description. A goal in this work is to show such a general method of dealing with a wide variety of data structures.

One of the general goals has been to provide explicitly as much of the structural information as possible about a composite data element. When all of the structural information is made explicit, then it can be changed during program execution, thereby providing for the maximum flexibility.

COMPOSITE DATA ELEMENTS DEFINED

The basic concept of a "composite". In the preceding section I have given examples of composite data elements and by these examples have established some terminology. Now I abstract the essence of those examples in order to develop the requirements for composite data elements.

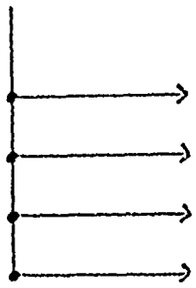
A "composite data element", or a "composite", for short, is a data element formed by an association or relationship of some number of other data elements. The "core", or "heart", of a composite is a set of "primary" component data elements which are to be considered as standing in some "primary" relationship to each other. This primary relationship can be thought of as a sequence of cells, where the contents of the cells indicate the primary component data elements. The sequence of the cells may matter, in which case we call it a "list", or it may not matter, in which case we call it a "set". (A cell does not necessarily correspond to a storage location in a computer).

Associated with this core of primary elements can be "secondary" data elements. I say that these secondary data elements stand in "secondary" relationships to the core. A secondary element may be related to a single member of the core, or it may be related to the whole core considered as a unit. We can think of these secondary data elements as conveying information "about" the core elements. This suggests the word "metadata" as a collective term for them. Thus we see that a composite is a complex network of relationships among data elements.

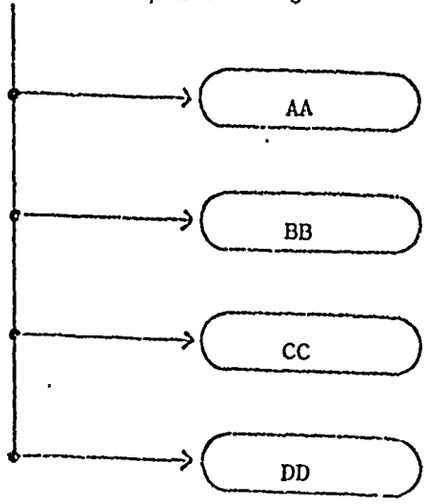
These primary and secondary associations are not required to be describable in terms of any simple storage concept for a machine. It would of course be convenient if it were so describable, for most programmers are trained to think in terms of storage concepts and manipulations. I foresee, however, that this concept of primary and secondary associations will not be representable in terms of present-day storage concepts in any neat way.

The fundamental need that has been illustrated is that of being able to associate a group of data elements. By "associate" I mean indicating what members are in the group, indicating whether the group is ordered or unordered, and being able to have this group play the role of a data element. By having a composite able to play the role of a data element, I mean that it can have identifiers and it can be a component of another composite data element.

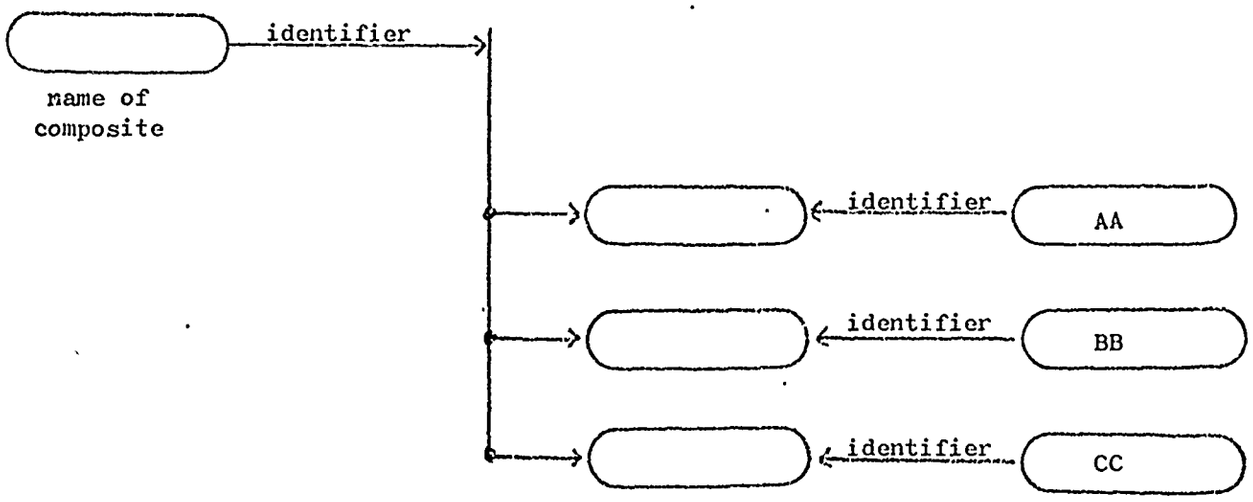
I am not concerned at present with how such an association is to be realized in a real environment. What I am concerned with is how the association appears from a conceptual standpoint. There are several possible views we might take: 1) that the association is represented in a list of the identifiers of the components, and 2) that the association is represented by some mechanism not know to the user, such as a list of the machine addresses of the components, and 3) that the association is a sequence of the actual components. #3 is untenable because of the fact that we may want the same set of components to appear in different composites in different physical sequences—an obvious impossibility. #1 and #2 differ only in that one uses identifiers (strings over the available alphabet) while the other uses machine addresses. The choice between these two is somewhat a matter of taste. In fact, they need not be mutually exclusive, as long as the processor can tell whether it is looking at a machine address or an identifier in a composite. #2 appears to me the better choice. I can graphically suggest it by the following sketch:



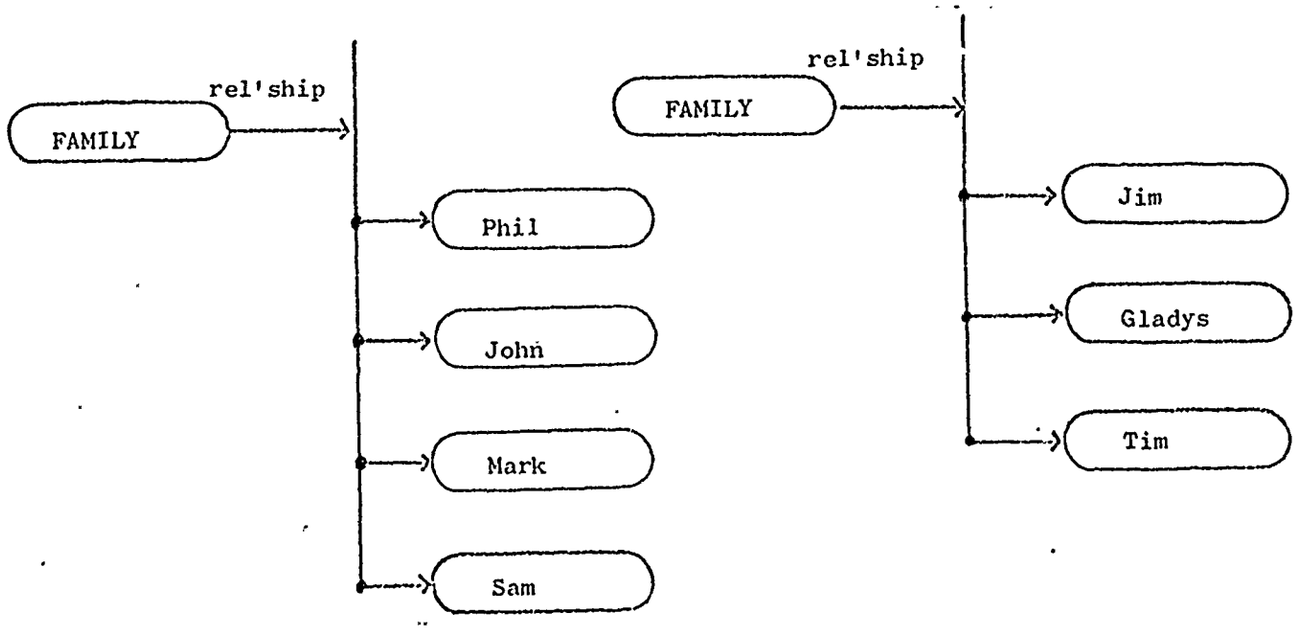
The pointers of a composite are invisible to the user. All the user has access to are the data elements to which the pointers point. If what we want as the composite itself is a list of identifiers (of data elements), we can easily provide for it, according to the following sketch:



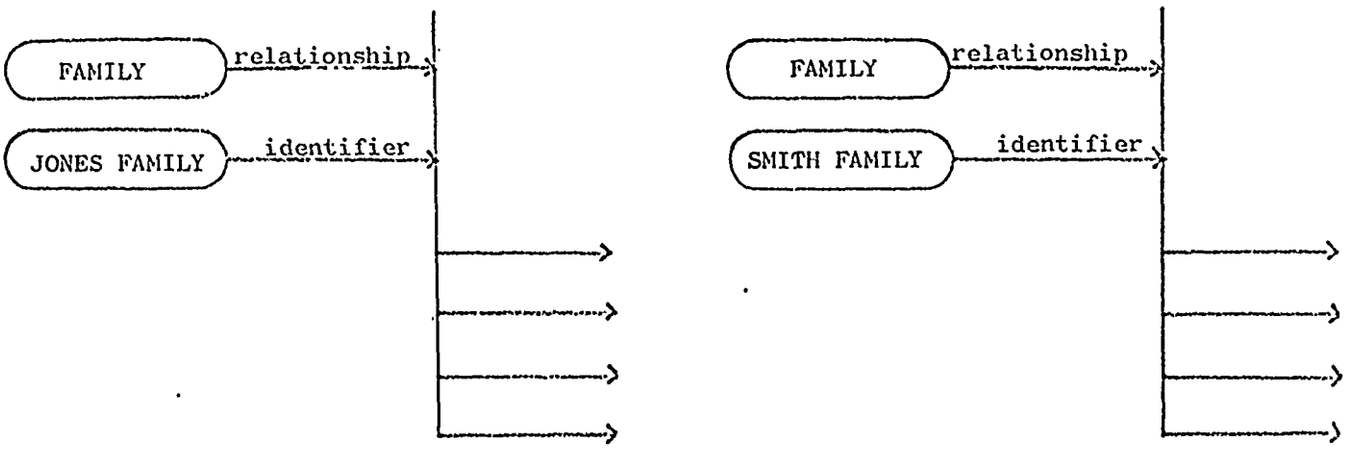
The pointers of the composite may point directly to a set of data elements, which elements may or may not have individual identifiers. We portray this latter situation, with individual identifiers, thus:



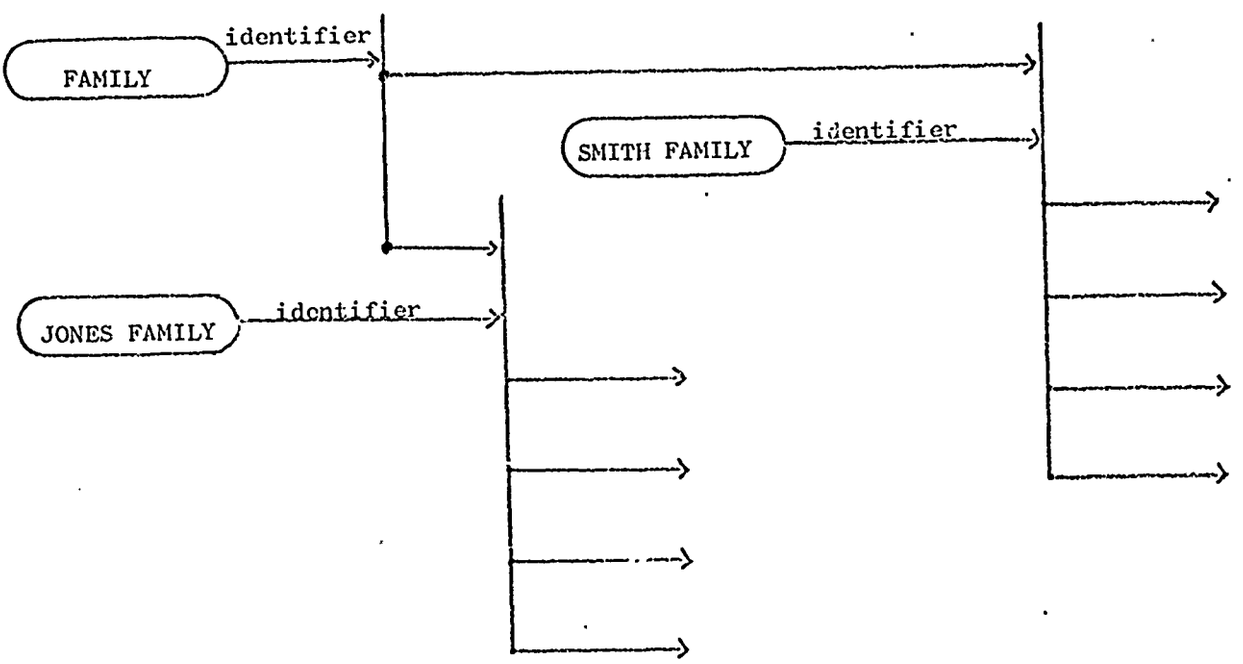
The "name of the composite" is subject to two interpretations. It can be the unique name of an individual composite, e.g., "JONES FAMILY," whose components are: "TOM", "MARY", "SUSIE", and "JACK". It can alternatively be the ambiguous name of a relationship, such as FAMILY, which is the identifier of all composites which portray family relationships. Then the following sketches depict two family relationships.



Of course, one can go a step further and have both individual identifiers plus ambiguous relationship names.



An alternative way of indicating relationship is by being in a given set. In this case the relation of FAMILY is the set of instances of family relationships:



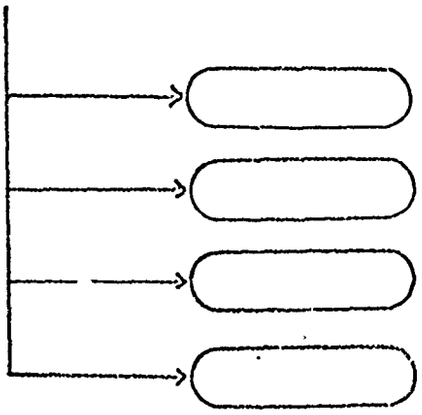
We simply note at this point that the concept of order versus lack of order in an association is a matter of some variation. It could be

left to functions defined on composites to consider them ordered or not. An indicator of order could be explicitly attached to each composite where it was desired to indicate specifically whether order was material; although in the majority of cases it might be left to convention rather than to specifically indicate it.

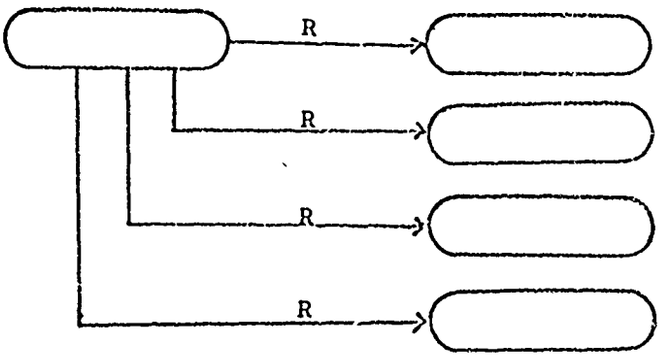
There are two fundamentally different meanings to the word "set" in the programming context:

1. an explicit set or sequence, expressed as a composite of components,
2. an implicit set, defined by a group of common characteristics.

The explicit set may be diagrammed:



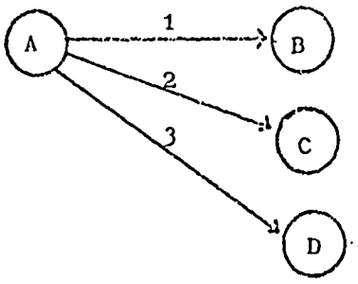
The implicit set may be diagrammed as follows, where the R stands for some relationship.



It is the burden of the user to know at all times which kind of set he is dealing with, because the transformations he uses may not apply equally to

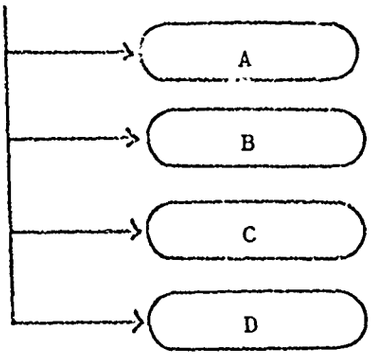
both kinds of sets.

By way of illustration, consider the task of expressing the following graph structure as a composite data element:

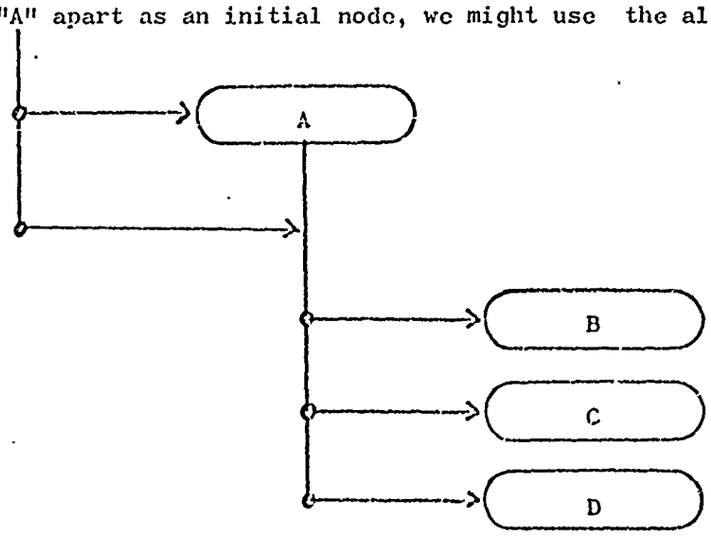


The nodes A,B,C, and D represent data elements not further defined. The arrows represent some homogeneous relationship, say,"potential succession." The numbers on the arrows represent some sequencing imposed on the relationship.

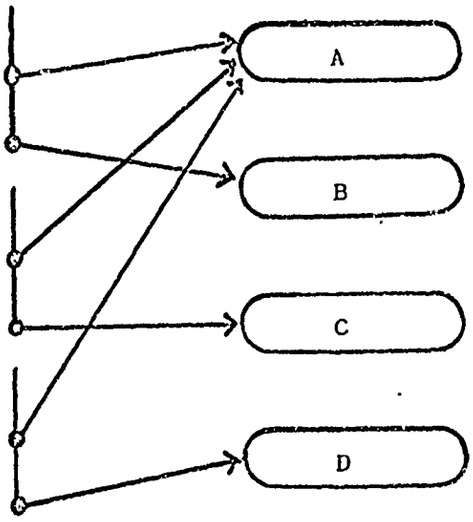
As long as the relationship indicated by the arrow is the same for all arrows, the relationship can be left "understood" rather than be made explicit. Thus a simple list could be used.



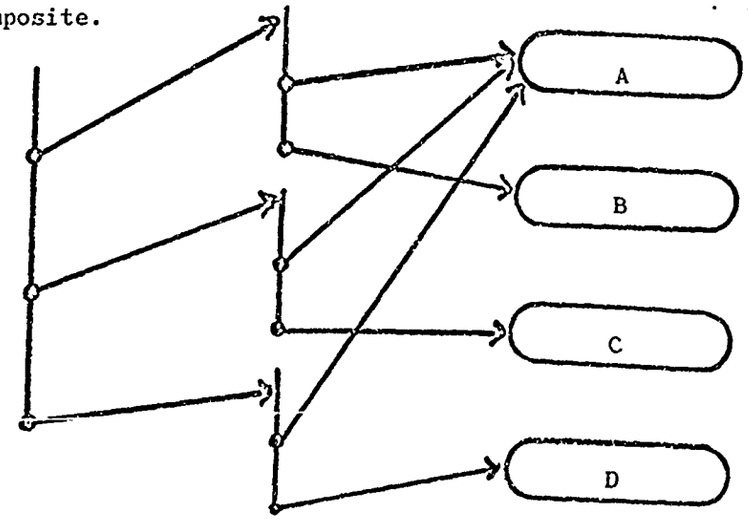
If we wished to set "A" apart as an initial node, we might use the alternative formulation:



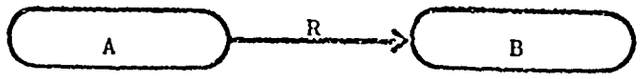
If we did not wish to consider the relationship a homogeneous one, however, or if we wished to attach one or more data elements (or metadata elements) to a relationship, each relationship could be expressed as a single composite, thus:



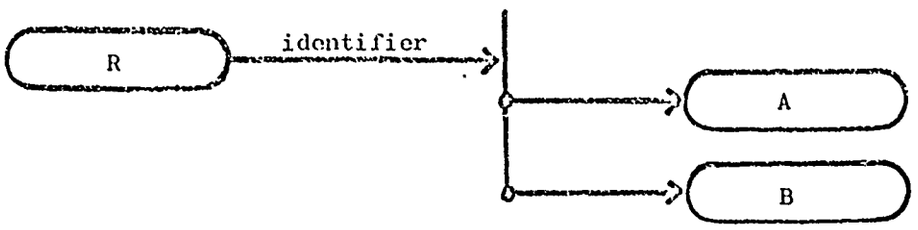
It is vitally important to note that here we have a set of relationships, but it is not an explicit set of relationships. An explicit set of relationships would be one in which the members, in this case relationships, were components of a composite. Here is an illustration of the same relationships as shown above, but here they are tied together as components of a single composite.



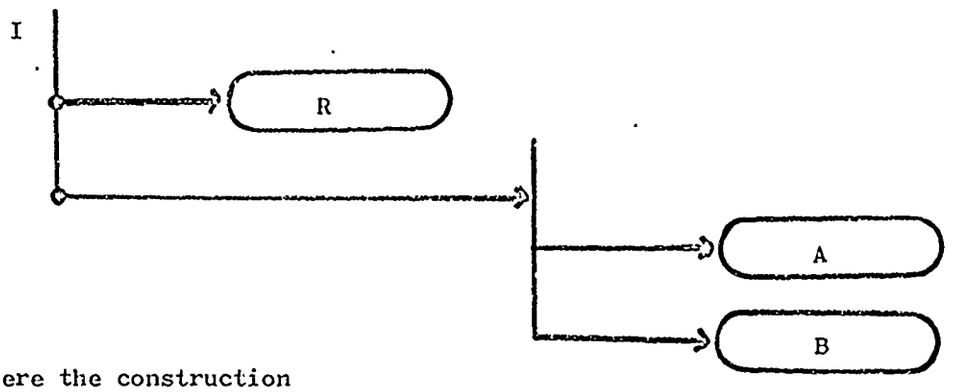
A two-element relationship has often been represented here by means of a single arrow labeled with the relation name, thus:



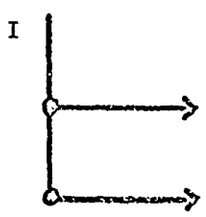
This is a shorthand notation for a two-component composite which is diagrammed as:



or even more properly diagrammed as:



where the construction



signifies the primitive relationship of identifier. The above shorthand notation will be used throughout this report.

Sets vs. lists. It is not obvious that it is really necessary to make an explicit distinction between a set and a list (that is, between an unordered set and an ordered set). We could in general leave undefined the property of being ordered or unordered. In specific applications we might want to label a composite as ordered or unordered: this can be done by relating a property of "order" or "unorder". When the ordering property is left undefined, then a composite is ordered or unordered according to the function that the user chooses to apply to it. That is, some functions may treat a composite as ordered, which it necessarily is because of the physical characteristics of machines. Other functions may treat a composite as unordered, and will not take advantage of the fact that

it is ordered in a machine.

What actions can be taken involving a composite. We can get a further understanding of a composite by noting the actions that can involve a composite during program execution:

1. We can obtain an identifier of a given composite considered as a unit. I say "an identifier" because there may be more than one identifier associated with the composite.
2. We can obtain an identifier of the nth primary component. Here we are talking about an identifier which applies to a primary component only in the context of the composite. I have in earlier examples called such identifiers "part-identifiers". Each primary component can have other identifiers unrelated to the existence of the composite.
3. We can obtain the value-string of a primary component, where that component is designated either by its ordinal position in an ordered list or by a part-identifier.
4. We can obtain the value-string of a secondary component, via expressing the binary relationship in which the secondary component stands to some identified primary component.
5. We can add secondary components without limit, and we can selectively delete them.
6. We can create a composite description, called a "model", which provides the information necessary to carry out actions 2,3,4, and 5 above for some class of composites.
7. We can create a new composite, and may give its structure in any of several ways:
   a. We can say that the new composite is "like" an existing one, in which case the structure and contents of the existing one will be copied. (This "like" feature has a similar counterpart in PL/I.)
   b. We can say that the new composite has as its model some existing model.
   c. We can say that the new composite has the form of a list or a set. Actually "list" and "set" can be regarded as primitive models.

The following sections describe composites in more detail.

Primary association. The first requirement of a composite data element is that it provide for an association of a set of data elements. I call this first association the "primary" association. It can associate any number of data elements. These associated elements can be simple data elements, composite data elements, or a mixture of the two. The associated set of elements can be either ordered or unordered. (I omit the partially-ordered case as being too complicated; it can be built up from ordered and unordered components.) Each composite must therefore have associated with it in some way, as yet unspecified, an indicator as to whether ordering is material.

The components of such an association may simultaneously be components of other associations. This suggests that, at least in such cases, the association of elements is not one of being physically proximate. It might be realized by putting in physical proximity some representatives (identifiers, pointers) of the elements to be associated.

Identifiers of composites. Having introduced the subject of composites, we are now ready to extend the notion of "identifier" to cover identifiers of composite data elements. Everything said earlier concerning identifiers of simple data elements also applies to identifiers of composite data elements. With the introduction of composites, however, the subject of identifiers becomes more complicated. The general concept of identifier relationships in a composite data element is introduced in Figure 4-5.

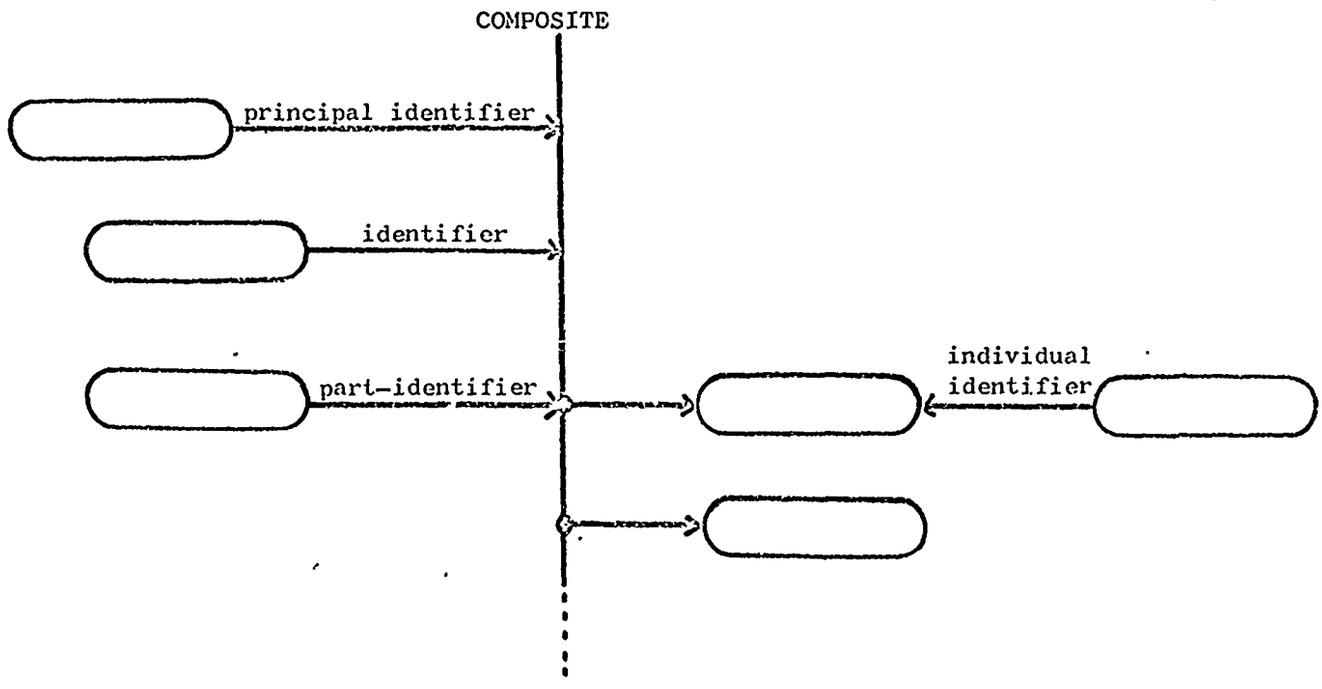


FIG. 4-5 GENERAL CONCEPT OF IDENTIFIER RELATIONSHIPS IN A COMPOSITE

Note that the part-identifier is in a sense a label on a position, while an individual identifier is a label on an element. However, probably the best way to distinguish a part-identifier from a position identifier is in the association transformation which, when a data element is moved, does or does not move the corresponding identifier.

Identifiers associated with data elements are one means of referencing those elements. When a data element, either simple or composite, becomes a component of some composite data element, it becomes accessible in another way which we say is "in the context of this composite:" we can refer to the component by a part-identifier. To show that a part-identifier is related to a given composite, we qualify the part-identifier by the identifier of the composite as a whole. Example: we might refer to FATHER OF JONES-FAMILY where "FATHER" is a part-identifier and "JONES-FAMILY" is an identifier of a composite. We recognize that this is an example of the hierarchical method illustrated below.

Note that the identifiers of parts can be attached in different ways. Consider the string

HE␣CHOSE␣IT

A substring of this string could be designated:

1. by character position, e.g., "characters 3 through 7 inclusive",
2. by characteristics or properties, e.g., "the first substring beginning 'CH' and ending in 'E'".

Both of these designations yield the same substring, namely "CHOSE". Let these substrings be named "A" and "B" respectively. For an invariant string it matters not which of these identifiers is used to designate "CHOSE".

In the case of a string which can be modified, however, it does make a difference how the identifier is attached. In the above string, substitute "SHE" for "HE". "B" names the same substring as before, "CHOSE". But "A" names the substring consisting of characters 3 through 7, which is "␣CHOS". This distinction suggests that we need two different varieties of the relationship "part-identifier of:" the first is "identifier of position", the second is "identifier of content".

The fact that a component of a composite can be referenced by a part-identifier has just been discussed. A component of a composite can also in general be referenced in ways other than by a part-identifier. One such technique is by giving its ordinal position. This makes sense only if the components are in fact ordered; that is, they are in a composite

which is a list rather than a set. Another technique of selection is to give some unique property; this technique will be discussed in more detail later.

Providing for identifiers for a composite as a whole and individual part-identifiers for its components does not exhaust the possible needs for identifiers. Consider the following case: a composite representing a set of children. There may be an identifier for the set as a whole, say "child-set-1". There will be identifiers for the individuals, say "Allen", "Barbara", and "Charles". We also need to provide for the notion of "child", which is a name for a single, undistinguished member of this set. There does not seem to be an intuitively nice way to handle this by another relation within the framework thus far developed. Rather than create a separate relationship for this concept, we can get the desired effect via a function which will select an arbitrary member of a named set. We need to be able to create a definition which says "'child' is any arbitrarily-selected member of 'child-set-1'".

Used by itself an ambiguous identifier will select a set of data elements. If it is necessary to select a single one of these elements, then other identifiers must be assigned such that some combination of identifiers will indicate the data element uniquely. There are two main ways in which such a conjunction of identifiers can be used:

1. Hierarchical, or ordered. The identifiers are given in hierarchical sequence; whether in sequence of "going up" or "going down" is understood by convention. Example: in COBOL we can have a compound identifier of the form "dataname-1 OF dataname-2 OF dataname-3", as in "DAY OF MONTH OF TAXABLE-YEAR". This means that MONTH must be defined as a component of TAXABLE-YEAR, and that DAY must be defined as a component of MONTH. DAY and MONTH may be names of components of other elements as well; that is, they may appear in other naming hierarchies, and these latter appearances will be independent of the former.
2. Set-intersecting, or unordered. The identifiers are given in any sequence. Each identifier names a class of elements. The intersection of the named classes need not yield a unique element, though under normal circumstances a single element would most likely be expected.

The problem of applying identifiers to points versus segments. There is a distinction between attaching an identifier to a point (for example, a starting-point in a series of imperatives) and attaching an identifier to a segment (for example, a set of imperatives constituting a procedure). If we say "performA" we expect A to identify a segment. If we say "Go to A", we expect A to identify a point. A "point" in a program or body of data corresponds to some data element; to refer to that point, we give an identifier of that element. A "segment" or "sequence" in a program or body of data is a set or list, which is a composite data element; to refer to that point, we give an identifier of that element. A "segment" or "sequence" in a program or body of data is a set or list, which is a composite data element; to refer to the segment or sequence as a whole, we give an identifier of the composite.

Metadata relationships. To any data element, or to any of the component cells of a composite, can be associated, in a binary relationship, certain data elements which represent data "about" the related element. We refer to such data as "metadata" and call the relationship one of "secondary association". Examples of metadata relationships are:

1. identifiers, "regular" and "principal", of whole data elements, or of their component cells (the latter heretofore called "part-identifiers);
2. reset, indicating the contents of the cell is to be reset to the value-string indicated by the contents of the metadata element;
3. cell prescriptor, which prescribes in terms of a domain/class prescriptor what is allowed as the contents of a component;
4. component prescriptor, which prescribes in terms of a domain/class prescriptor what is allowed as the contents of a component designated by the contents of a cell;
5. access code, which can limit the type of access and accessor to a composite or to a component designated by the cell contents;
6. repetition factor, which when used in a model tells how many occurrences may exist of a given class of component;
7. uniqueness indicator, which can be used to indicate an element which can be modified without fear of upsetting other relationships.

Some of these metadata concepts are illustrated in Figures 4-6, 4-7, 4-8, and 4-9. The user need not be limited to the set of metadata relationships

which are presented here. He should be able to define other relationships when he needs them.

Figure 4-6 conveys the following facts: There is a composite of which "quantity" is the identifier of one of its components. The simple data element representing the "quantity" has the individual identifier "A", but as yet has no value-string assigned. When a value-string is assigned, the component prescriptor "integer" specifies that it must be from the domain of integers.

Figure 4-7 co veys the same facts with the exception that the existence of the composite and a part-identifier is not mentioned.

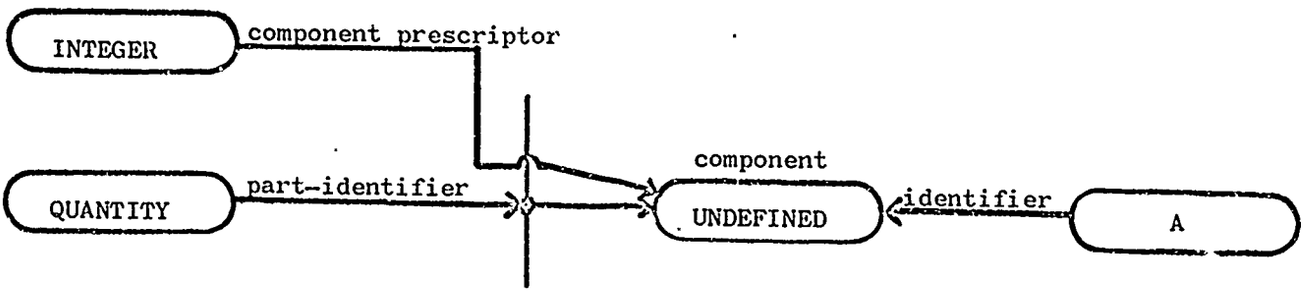


FIGURE 4-6.

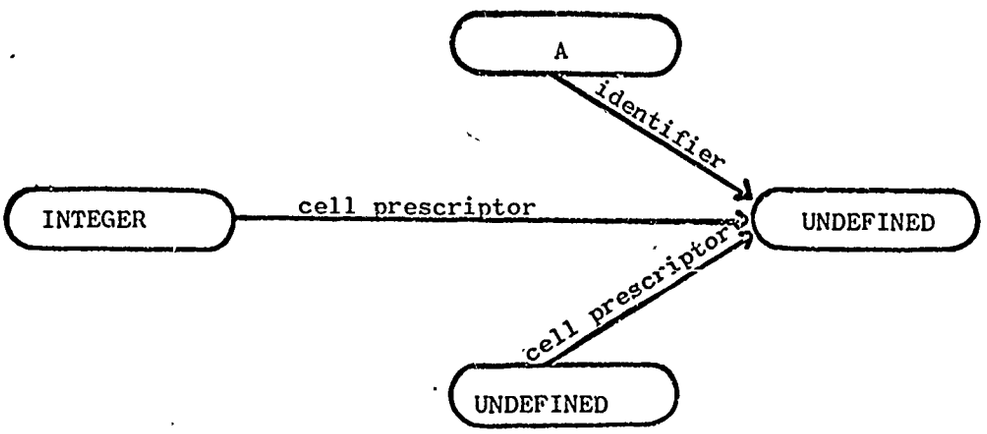


FIGURE 4-7.

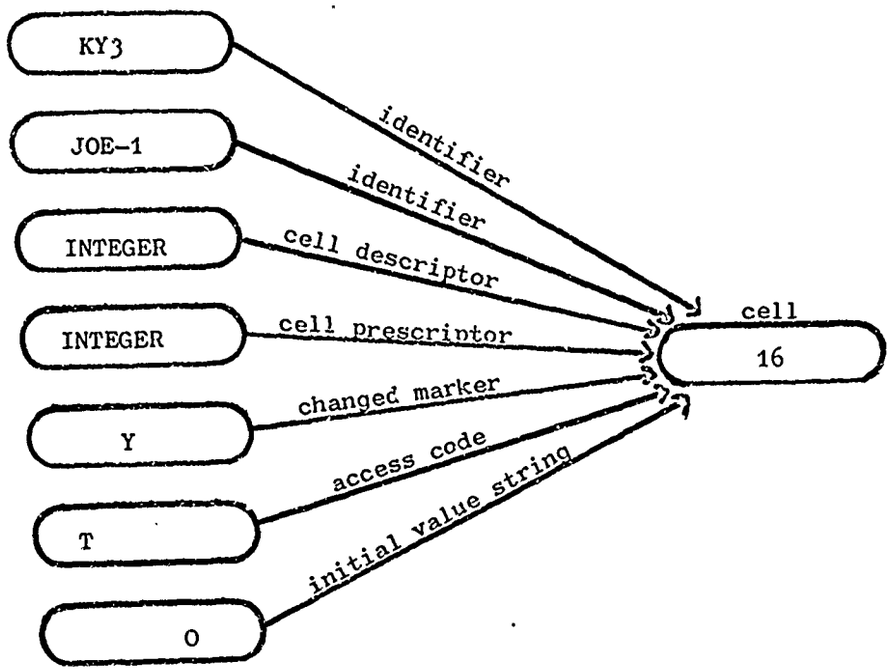


Figure 4-8. Examples of Metadata Associated with Cells.

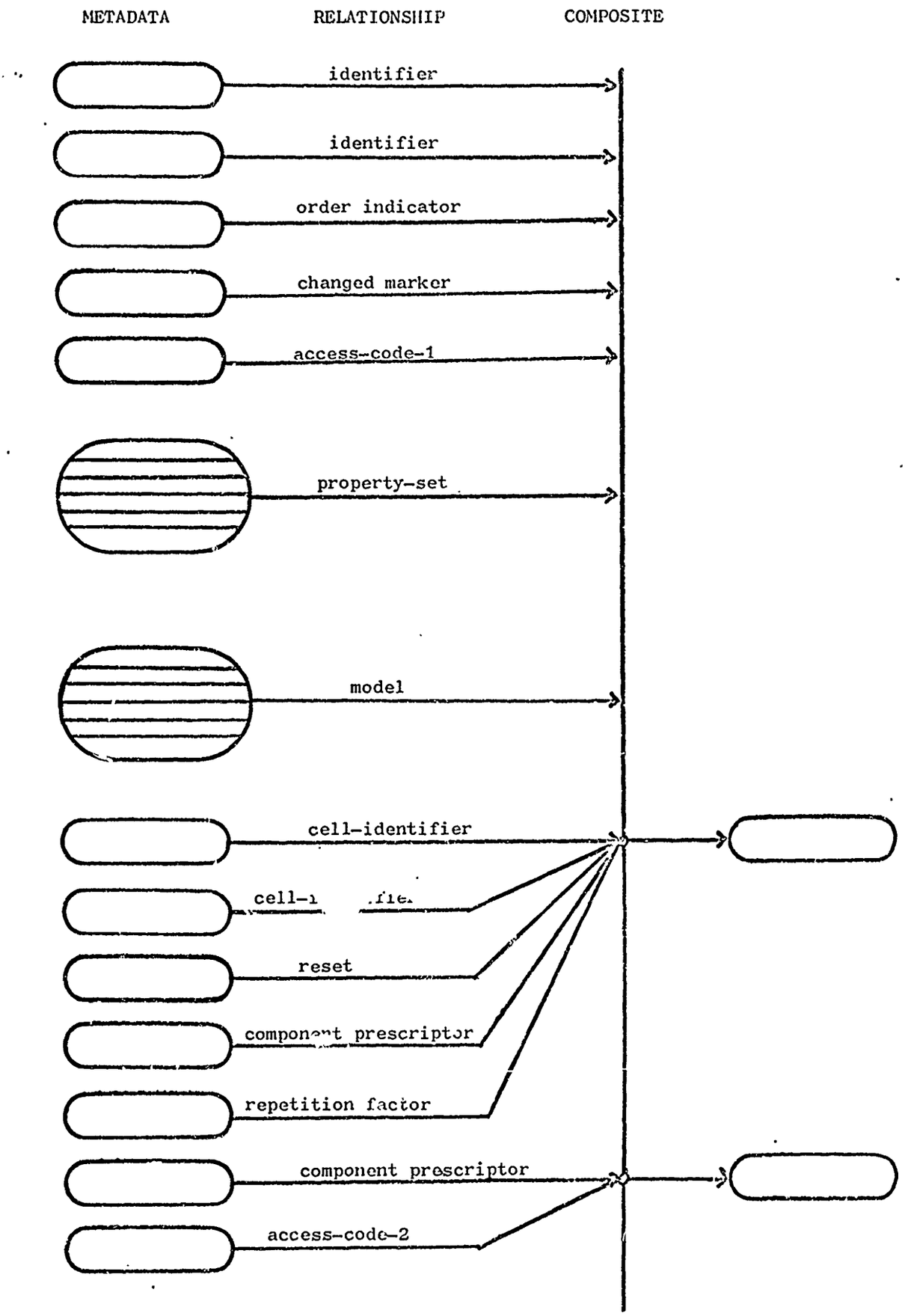


Figure 4-9. Example of a Composite Data Element and Associated Metadata

It should be noted that an element which stands in some metadata relationship to another element is in some sense not a "normal" data element. Many of its characteristics are implied by the type of relationship. For example, the element which stands in the domain-prescriptor relationship to some other element need not itself have an associated domain prescriptor; the domain-prescriptor relationship itself serves to imply that such an element can only have value-strings from the domain of domain-identifiers.

A somewhat confusing issue is the fact that some relationships apply between a metadata element and the contents of a cell, while others apply between a metadata element and the element designated by the contents of a cell. This will become clearer in the pictorial illustrations given later under "structures of a composite".

We can now make the general observation that any element which plays a metadata role need not have metadata elements tied to it. The type of relationship of the metadata element to the data element implies what the domain of the metadata is. This is not to say that a metadata element cannot in turn have a metadata element; but it need not have it. The requirement for metadata must obviously terminate at some level, and we have chosen the lowest level as being the most convenient place to terminate it.

We now proceed to discuss these metadata relationships in more detail. The most important of these is the identifier relationship, which has already been discussed above.

"Reset" concerns a relationship used for resetting or initialization upon command. Resetting of a cell, putting a new value in a cell.

A "cell prescriptor" is used to define from what domain or domains of values may be taken the value-string for the related element, presumed to be a simple data element. Thus a cell prescriptor may be the name of a domain, such as INTEGER, or it may be the name of a set of domains, such as NUMBER, elsewhere defined to mean "INTEGER OR RATIONAL".

A "component prescriptor" is used to define from wha class or classes may be taken the data element which corresponds to that component. Thus a component prescriptor, as a metadata element, can contain the name of a class, such as FAMILY, or it can contain the name of a set of classes, such as STOCKHOLDER, elsewhere defined to mean "PERSON OR COMPANY". It is

quite possible for a component name and its associated component prescriptor to have the same identifier. An example is "month" in the illustration of the composite element for "clocktime". The distinction between cell prescriptor and component prescriptor is shown in Figure 4-10. The cell prescriptor and the component prescriptor are alternative ways of giving the same information about the domain of the contents of the principal value cells.

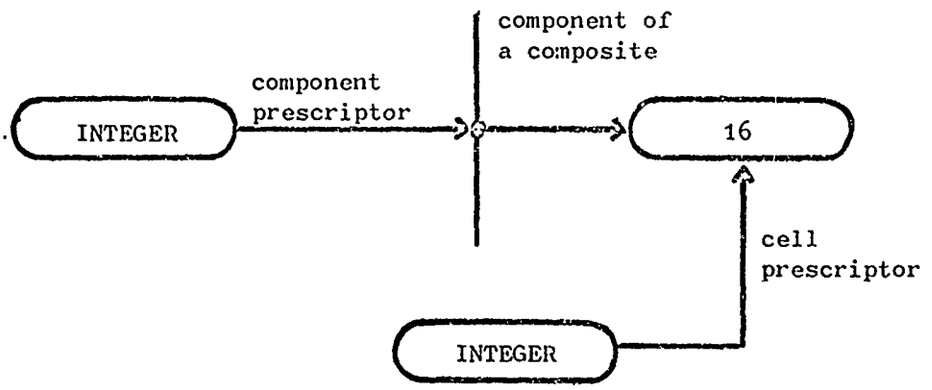


FIGURE 4-10. DISTINCTION BETWEEN COMPONENT PRESCRIPTOR AND CELL PRESCRIPTOR

An access code can be used to limit the access to the related data element. It may limit the type of access, to READ ONLY. for example. It may limit the accessors, by allowing access only to those users presenting a specified code.

In a model, to be discussed in further detail later, is presented information about a class of similar composites. Certain composites may be variable: that is, a given component may be allowed to occur multiple times. A repetition factor in a model can be used to specify the numbers of occurrences of a given type of component in any composite. It may give a minimum value, a maximum value, both a minimum and a maximum, or it may give some set of allowable values.

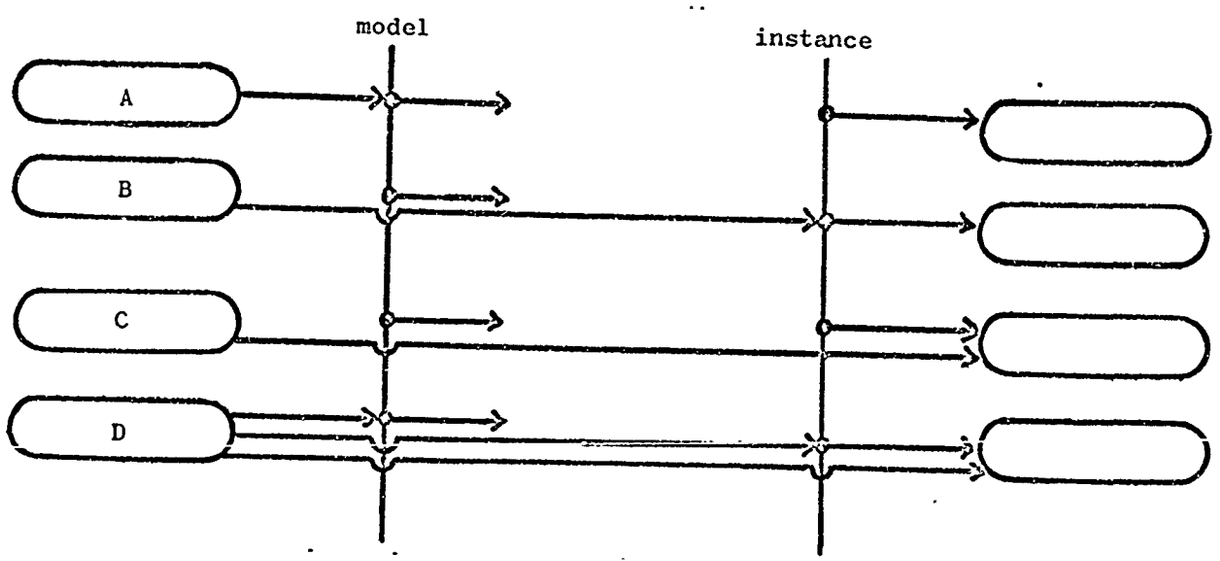
A "uniqueness" indicator could be used to indicate that some element (data or metadata) was to stand in some specified relationship to only one other data element. This would be used to show that the latter could be modified without inadvertently spoiling another relationship. Example: If A stands in relationship R_1 to J and A stands in relationship R_2 to J and if the user wishes to modify that element which stands in relationship

R_1 to J (namely A), he runs the risk that the relationship A R_2 J is no longer valid, even though it is still explicitly expressed in the data structure. If the user knows, by a uniqueness indicator associated with A, that A does not stand in any relationship with any element other than J, then A can safely be modified.

The concept of a model I feel is an important one. We have several possible courses of action on how a model might be realized:

1. It could be a primitive concept, not defined in the language, in which case certain transformations for manipulating models would need to be provided and learned;
2. It could be a composite in the form of a standard structure, with some interpretation rules built into the processor.
3. It could be left to the user to construct his own models and provide his own accessing transformations.

We may want part-identifiers associated with the model rather than with the individual composites which are instances of the model. This brings to four the number of ways that an identifier can be related to a component of a composite(which is an instance of a model). These ways are illustrated in the figure below.



These four ways are:

1. To relate the part-identifier to a node point of the model, as with "A" in the figure;
2. To relate the part-identifier to a node point of the instance, as with "B" above;

3. To relate the identifier to the corresponding cell of the instance, as with "C" above;
4. To relate the identifier simultaneously in all three ways just mentioned, as with "D" above.

All but one of these ways may seem intuitively objectionable because of the fact that an identifier is not tied directly to the component it identifies. How the component is in fact located given the identifier, or how the identifier may be located given the component, is an implementation matter which need not concern the user.

Models. There are two fundamentally different approaches to expressing composite data elements. The first of these approaches is to have a composite "carry its own" descriptive information (metadata), in the sense of having these metadata elements directly associated with the components. The second of these approaches is to collect in one place, called a "model", some or all of the descriptive information common to a class of composites.

We illustrate these approaches. Consider the concept of human family. Here are two self-describing, or "modelless", instances of family (where square brackets set off information which is understood but not explicit):

[Instance 1]		[Instance 2]	
[Identifier:] F1		[Identifier:] F2	
[Part-Identifier]	[Value-String]	[Part-Identifier]	[Value-String]
father	Harry	father	Sam
mother	Susan	mother	Molly
children	{Betty, Margaret}	children	{Michael, Stephen, Alice}

It may be convenient, however, to abstract the common information from these instances and place it in another data element which we call a "model":

[Model]	[Instance 1]	[Instance 2]
[Identifier:] FAMILY	[Identifier:] F1	[Identifier:] F2
[Part-Identifier]	[Value-String]	[Value-String]
modelname	FAMILY	FAMILY
father	Harry	Sam
mother	Susan	Molly
children	{Betty, Margaret}	{Michael, Stephen Alice}

Note that in this latter illustration, which we called the "fixed model" case, elements (here simply part-identifiers) of the model must stand in

1-to-1 correspondence with the elements of each instance. In this illustration the only common information is part-identifiers. Other kinds of common information, however, could be included: anything in the category of metadata, described earlier.

Briefly then, a model contains metadata abstracted from a class of composites, plus possibly some information about the structure of the composites, such as the numbers of repetitions of components. I originally thought of a model as information on how to interpret an instance. This is obviously not the case. A model is that descriptive information (metadata) abstracted from a class of instances; or in the case where no instances have yet been created, the information in the model may be regarded as a prescriptor.

A model, it should be noted, is not a prototype, or "skeleton", the concept of which will be explained later.

It is appropriate to use a model in connection with a simple data element, which can be considered to be a composite with only one component.

One kind of information provided by a model is identifiers of components. For example, a model might be used for a class of data elements called "complex numbers," each member of the class being a pair of numbers, the first member being called "realpart" and the second member "imagpart".

Notice that a part-identifier names the individual members of a class of components. For example, "realpart" names, ambiguously, each first component of all the component pairs representing complex numbers. Only when one of these part-identifiers is qualified by the identifier of a particular composite is the ambiguity resolved.

The information which can occur in a model can be given in any of several ways. A given metadata element may be given as a quoted value-string, as an identifier ultimately interpretable as a value-string, or as an indicator that the value is to be taken from a given component of the corresponding composite. This latter means that the number of repetitions of some component, for example, might be given as another component of the same composite.

Observe that while a model is capable of carrying information common to a set of composite data elements, not all of such common information need be carried in model. Which data is carried in a model and which in the composites themselves is up to the choice of the user. His choice will depend on the ways that he will access and modify information.

The information in the model must be recorded in such a way that it can be matched up with the parts of the composite to which it applies. The simplest way to achieve this correspondence is to have a fixed composite and a fixed model; then the correspondence is simply 1-to-1. However, we want more flexibility than a fixed composite gives us. We want the possibility of having composites which can have components added or deleted dynamically. If we wish a model to describe a set of such variable composites, then clearly the simple 1-to-1 correspondence won't work. The model must carry enough more information to be able to set up a correspondence between the components of the model and the components of any related composite. Such a model we call a "variable model". One of the kinds of information unique to a variable model is "repetition factor" which tells how many of the corresponding components we may expect to find in a composite. We may establish some conventions, too, such as the convention that an omitted repetition factor is taken to have the value "1". A variable model should also provide a way to state under what conditions a given substructure is to occur. Obviously at this point we have left a great deal unsolved or unsaid about variable models.

To every instance of a model, together with the model, there exists a corresponding non-modelled, or modelless, or self-contained, instance. These two forms are equivalent in one sense, but not equivalent in a second sense. In terms of the information contained, and from the point of view of accessing the instance, they are equivalent. From the point of view of accessing the model, or of modification of any information other than the value-strings of the instance, they are not equivalent. For example, consider the modification of a part-identifier. If the part-identifier is associated with the component of the instance, the part-identifier change affects only that instance. If the part-identifier is associated with the component of the model, then the change affects the corresponding components for all of the instances of that model.

We note also that the modelled and modelless modes of representation are not mutually exclusive. They can be mixed. At any level of detail a model can be used within a modelless framework, and vice versa. Examples: a document can be described using the model framework, while the publication agency for the document uses the modelless framework. Conversely, a document description can use the modelless framework while the publication agency is handled in a model. It is a search for "right mixture" of

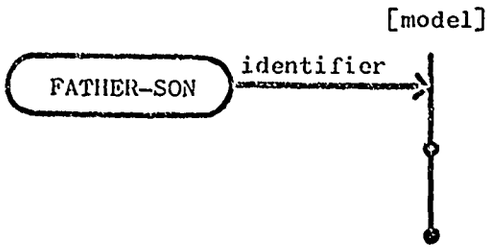
modelled and modelless concepts which is one of the time-consuming occupations of the systems analyst. He is always trying to abstract from instances as much as possible which fits into a regular structure, for it is easier to describe and more economical to manipulate.

It is generally the case that a given programming language provides for either the modelled or the modelless structures but not both. I consider it a critically important fact that experimental applications demand at least the flexibility afforded by the modelless structure; I therefore believe that languages which don't provide modelless structures are unfit for any non-production type application. Full flexibility of course demands that a language provide for both the modelled and modelless structures.

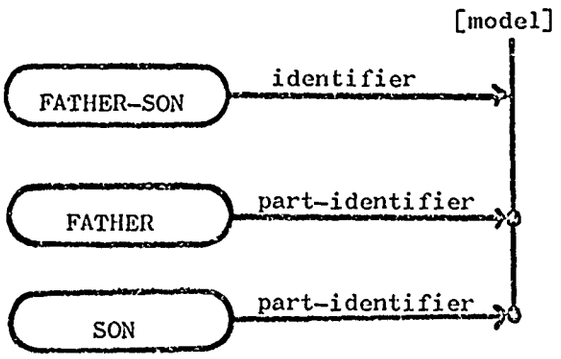
Standish, in his dissertation on data definitions proposes only a model concept, similar to that outlined above, for describing composite data elements. It seems to me that it is essential to have the modelless instance concept. Imagine for the moment a model and a sizeable set of instances of a composite data element type representing library books. Suppose that we wish to record the fact that two specific books in the library have been lost, plus the date that the loss was discovered; further suppose that no provision for recording this fact exists in the composite data element as defined. In order to record this fact, it is technically possible, but manifestly undesirable, to modify the model, with the consequence that every instance must be modified to conform to the new structure. One would rather "tack on" to the two instances in question some expression of the desired fact. But in doing so one would create non-standard instances: that is, they would no longer wholly correspond to the model.

Skeletons. A skeleton is an element related to a class of elements, such that every element of the class is initially a copy of the skeleton. An instance of the class is first created by copying the skeleton, including possibly some initial values in its components. After this initial creation, of course, each instance can be modified in an unrestricted way. A skeleton looks like an instance, with its value-strings undefined or set to initial values. An instance generated from a skeleton may or may not give the name of the skeleton from which it was derived, depending on the wishes of the user.

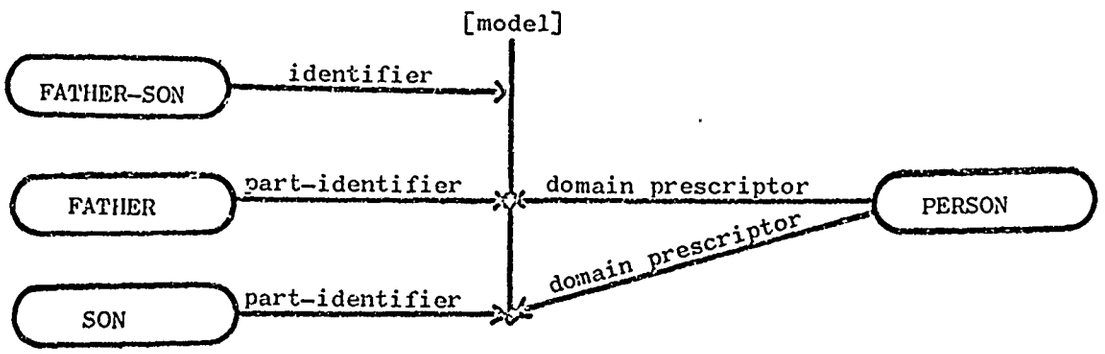
A simple example of a composite, its model, and some results. The following example may help to clarify the notion of model and composite. Construct a two-element relation, or ordered model, called "FATHER-SON".



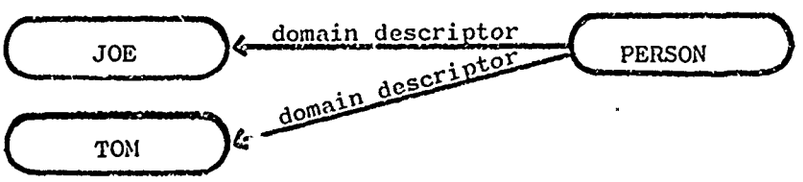
Its part-identifiers are FATHER and SON, respectively.



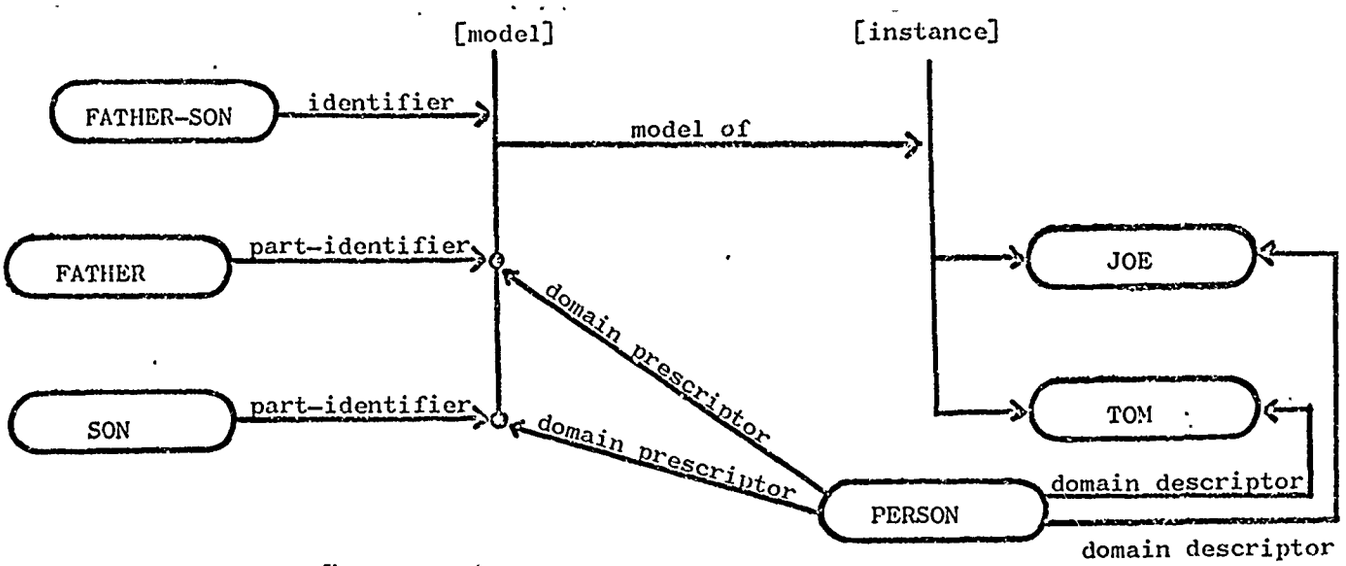
The components are prescribed to be from the domain of PERSON.



JOE and TOM are simple data elements from the domain of PERSON.



JOE and TOM stand in the relationship of FATHER-SON, where JOE is FATHER and TOM is SON. ("The relationship" of FATHER-SON is an instance of the "relation" FATHER-SON.)



Value of a composite. It is interesting to discuss in what sense a composite data element can be said to "have a value". In the case of a composite data element whose components are values as expressed by value-strings, the set of values possessed by that composite element could be said to be a value (a "vector") from a Cartesian product space. Where a component has as its "value" not a value-string but rather another composite data element, the value could be regarded as one coming from a Cartesian product space where some factors are not sets of individuals, but sets of vectors. Even the latter viewpoint may not be particularly sensible, for these sets of vectors need not be of uniform composition, such as the set [(0), (1,2), (3,4,5)].

Properties, property-sets, and property-lists. There are fundamentally different ways of treating the expression of properties: the "unary-relation" viewpoint, and the"property-value" viewpoint. Under the unary-relation viewpoint, a property is expressed by a set of names of the elements having a given property. For example, to express that an element has a certain property, say, "is red", or "is 5 inches long", we say that the property belongs to the set of elements having the property "is red", or "is 5 inches long". Under the property-value viewpoint, to express that an element is red, we associate with the property-name "color" the value "red"; similarly, to express that an element is 5 inches long, we associate

with the property-name "length in inches" the value "5". An alternate to the last example is to associate with the property-name "length" the value "5 inches". Both of these viewpoints have their merits, and it is inappropriate to arbitrarily rule out the use of either one. I admit, however, to a strong preference for the second method; most of the discussion which follows is concerned with the property-name-and-value viewpoint.

Properties of an object (in this context, properties of a data element) can be expressed in either of two ways: implicitly or explicitly. Properties expressed implicitly are called implicit properties. Implicit properties are those whose values are discoverable only by search or by algorithm: they are not explicitly given in some "property-list". Examples of implicit properties are: number of components in a composite, minimum value of a given component (that is, a component having a given part-identifier, for some set of composites), whether a given access code lies within a specified range, whether a designated value-string equals a given value-string. Explicit properties, on the other hand, are those that are explicitly given in some way. An explicit property may take the form of:

1. a component of a composite;
2. a metadata element standing in a specified relationship to a data element;
3. a component of a particular composite called a "property set", which is a composite standing in a property relationship to a given data element. Explicit properties are distinguished from implicit ones by being capable of being looked up directly, of being found by a simple search of a given property set. He who would use property values must know whether they are implicit or explicit, because the method of modifying them depends on in which category they fall.

Note that a property set, in any of its possible forms, can be associated with:

1. a simple data element;
2. an instance of a composite data element;
3. a special case of (2) above: a composite (representing a set of homogeneous data elements), implying that the properties apply to the members of the set considered <u>collectively</u>;
4. a model of a composite data element, implying that the properties apply to the instances considered <u>individually</u>.

Consider for a moment the case where a property-set is expressed in the form of a composite tied in the property-relationship to a data element. It is interesting to note that this property set (or list) can be expressed either as a modelled or as a modelless composite.

Property values may be either value-strings or names of data elements. For example, a set of composite data elements may correspond to a set of books. Some or all of these data elements may have an associated property named "color of binding" whose corresponding value-string is from the domain of colors. Some of the composite data elements may have a property named "publisher" whose corresponding contents or "value" is the name (identifier) of a data element which represents a specific publisher. Such a data element representing a publisher might, for example, have as components the publisher's name, address, type of publications, and standard discounts.

Certain useful properties apply to the representation rather than to the thing represented. Examples of such properties are: access-control-indicator (which might take on such values as "privileged access only", "read only", "initialize only"), and changed-indicator (which might take on such values as "unchanged since reset," and "changed since reset").

It is appropriate to raise the question: what is the difference between a property of some element and a component of that element? Put more concretely: if a data element exists to represent a book what aspects or attributes of this book are components of the composite representing the book and which attributes ought to be on some property-list or in some property-set related to this composite? I think this is a matter of viewpoint, or of taste. Some programming languages seem to let an object be represented by a list of its properties and their values. For certain applications it probably does not matter whether these properties appear as components or on a separate property list. One example will serve to indicate that there might be some point in making a careful distinction. Consider a family unit represented as a composite, where the components are "father", "mother," and "children". This family unit may have properties such as "address" and "religion". It would be misleading, however, to make these components of the composite "family". It makes more sense to have them on an associated property-list (or in an associated property-set).

We note in passing that the languages and system AEDNET represent

some signal must be provided to indicate that the modified interpreter is to be used. A special flag position might be reserved in every data element for this purpose, or a special metadata item might be used as a flag.

Another related capability is the ability to define new metadata relationships. The user must be able to give the new relationship an identifier, and define how the metadata item is to be treated when accessed (that is, whether it is to be evaluated or to be copied).

Input and output of composites. The expression of composite data elements, for the purposes of input and output, is a matter quite separate from the structure of these data elements inside the machine. Considered statically (not from the viewpoint of incremental change) the modelled and modelless concepts are equivalent: hence the input and output of data structures can be based on either point of view. It is up to the input-output routines to provide whatever conversion may be necessary. It is generally easier for purposes of input, for example, to organize the data elements of a composite in some tabular form, paralleling in some sense a modelled format; the actual resulting structure desired in the machine may be the modelless form. It isn't yet clear whether data elements as a whole should have some convenient notation, or whether we should be content to express in any one expression simply pieces of a data element. What expressions one uses is primarily a matter of choosing a notation, which matter we postpone until later.

Relationships versus composites. We call an element of a relation-set (which is an explicit relation) a "relationship". A fundamental question is whether such a relationship is essentially different from a composite. Both a relationship and a composite are an association, possibly ordered, of a set of components. One or more part-identifiers may be associated with each component. The association as a whole may have one or more identifiers. A set of similar associations may in turn belong to a set which is itself a composite.

When a relation-set is considered as a composite, the identifier of the relation is the identifier of the relation-set. The individual relationships do not need to take identifiers; they are usually referred to as being some (undesignated) member of the relation-set. When it comes to specifying members of relationships, a binary relationship is rather unique; it is easy to say "the other element of the relationship

elements by their property-lists, and represent composites by tying a set of elements together with a ring of pointers.

Manipulation of composites. For creating, accessing, and modifying composite data elements I postulate the existence of a "standard" data interpreter. We in effect enter this interpreter with a message as to what action is desired, and the interpreter provides the necessary action. The data interpreter provides for the storage of the data, so that the physical and logical problems of storage are hidden from the user.

Listed here are the abilities which are wanted in connection with processing composite data elements; it is the providing of these capabilities that is the job of the data interpreter:

1. To determine if the sequence of components matters;
2. To obtain the components separately, and in sequence if sequence is material;
3. To associate identifiers with the components, and to find a component given its identifier, or to find an identifier given a component identified by some other means;
4. To access or modify the "value-string" of a component (assuming it is a simple data element), and to find the domain of this value-string;
5. To access or modify any metadata standing in a specified relationship to a specified component, such as domain prescriptor, component prescriptor, access code;
6. To determine if the composite has a model or a property set, and if so, to access and modify its parts in the same way as can be done for a modelless instance of a composite;
7. To be able to add components to, or delete components from, or resequence the components of, a composite or its model;

In summary, to be able to create, access, and modify structures and values of composites and their models.

The matter does not end here, however. Another needed capability is that the user be able to modify the interpreter to provide some feature which he needs. This in turn brings up two matters. First, there must be some prescribed way to modify the interpreter; just how this might be done must be postponed for later consideration. Second, there must be some way to indicate when the modified interpreter is to be called into play;

in the relation R in which the element E occurs" With n-ary relationships, with n>2, we must say analogously: "the component in role k (or in ordinal position k) in the relationship (composite) of relation R that has element E with part-identifier I".

The expression and testing of relations. There are two approaches to the expression of relations:

1. The explicit method. This utilizes an explicit relation list or set, which is a set of n-tuples of the identifiers of the elements which stand in the relation. Logicians call this the "extensional method". I call such a set a "relation-set".
2. The implicit method. This is based on an algorithm which decides whether a given set of elements stand in a given relation. That is, it returns a value of "true" or "false" given an n-tuple of identifiers. Logicians call this the "intensional method". I call such an algorithm a "relation-algorithm".

The explicit method is advantageous where additions are to be made to, and deletions are to be made from, the relation-list. The predicate (the question of whether a given set of elements stand in the given relation) is made by a search of the set of n-tuples. This latter operation is relatively easy in an associative memory device. The implicit method utilizes no relation-set and therefore does not lend itself readily to applications which require additions to and deletions from the relation, because such changes require reworking of the algorithm. Naturally the algorithm constitutes the predicate. This implicit method is preferred for relations on large sets, particularly where the basic elements are in some metric (have a measure of distance or ordering). An example would be the "greater-than" relation on a set of integers.

Because of the inherent differences in these two techniques for dealing with relations, the user must know which method (the explicit or the implicit one) is being used in any given circumstance.

Properties of relations. A relation may have properties. such as being transitive, or not, and being commutative, or not. These properties can be implicit (discoverable only by examination) or explicit (expressed in a property-set associated with the relation).

Defining a domain: continued. Now that we have considered the concept of composite data element we are ready to return to the issue of defining a domain. Recall that a domain of value-strings is to be a set, possibly ordered, which can be added to, and manipulated, by the user.

A domain is also used by the processor for checking whether a given domain contains a given member and for whether a given member stands in a given relation (possibly of ordering or of equivalence) to another given member. The composite data element is an appropriate device to meet these needs, in those cases where a domain is expressible by enumeration, or listing, of its members. An explicit domain is then a set, or an ordered set, of strings. In order to be able to construct such a composite, however, the domain of strings must already have been defined; this becomes a requirement on the virgin system.

Orderings. Let us review the needs for various kinds of orderings. First we may have orderings based on a single relation, which we might call "precedence" These include:

1. Unspecified ordering, the usual meaning of the word "set"; examples of unordered sets include relations and functions.
2. Partially-specified order, expressible by a combination of lists and sets; examples of partially-ordered sets include lattices and trees, list structures and algorithms.
3. Wholly-specified order, corresponding to, and expressible in the form of, a list, examples include strings, iteration lists, parameter lists, and vectors.
4. Multiply-specified order, expressible by a set of lists; examples include arrays of dimension greater than 1, and multi-listed records (as in the Multi-list concept).

Where a single relation is involved, it is often not expressed explicitly, but rather is left "understood". When the relation involved is not homogeneous, however, (that is, when there is more than one relation involved), then simple sets and lists are no longer sufficient to express the ordering; we must instead go to a more complex structure: namely, the network, or association, concept.

Defining orderings. There is a problem of defining orderings, both on domains and on data elements. Such definitions are needed as a basis for generalized transformations which put data elements in sequence, or which test that a set of data elements is in sequence. We need a general method of specifying orderings. There are several possibilities: one is to let the method of specifying order be undefined in the system, and to have primitives which allow the specification and testing of order. A second possibility is to define orderings by data structures. This latter

seems the more flexible. It has the advantage that we are not limited to the kinds of order that one might build into the primitives but can express an unlimited variety of orderings. One might, for example, wish to express partial ordering.

Elsewhere is discussed the usefulness of orderings expressible as data structures for the purposes of control sequencing.

It is tempting to think of having ordering of elements in a domain (such as collating sequence) be a primitive concept whose method of representation is undefined in the system. The advantage of this viewpoint is that a processor can be arranged to create and test such orderings more efficiently than if the orderings are expressed as data structures. However, I have committed myself not to be swayed by concern for efficiency of implementation.

Furthermore, we must not lose sight of the fact that ordering is a relationship. Hence it has alternate modes of expression: by relationship indicators between cells (which means that the members must be explicitly stored in a data structure), and by algorithm.

Note that there are two important types of ordering or precedence relations. The first is the familiar ordering relation, which is transitive. "Greater-than" is an example of this type of relation. The second type is the "immediate precedence" relation, which is not transitive. Example of this second type: "is to the immediate right of", "has a forward connection to". The importance of this second type of relation is in the formation of loops. In a loop, a circular element, the transitive type of ordering relation cannot be used; it would lead to a contradiction.

Defining lexicographical ordering. It is important for some applications to have transformations which create lexicographical ordering and which test for it. In order to have these transformations, we must have some way to define lexicographical ordering.

The concept of lexicographical ordering is one of ordering strings of characters, where the strings have been made of equal length by padding on the right (or left) end with blanks (or zeros). An ordering ("collating sequence") must be defined on each character position in these strings; usually this collating sequence is the same for all positions. Examples of simple lexicographical orderings are: sequences of decimal integers, sequences of mixed radix numbers, alphabetized lists of English words.

This lexicographical ordering concept can be extended to super-lexicographical ordering wherein each position contains a string rather than a single character. The set of strings permissible in a given position is defined. This set must have some ordering imposed on it; this ordering could be, but need not be, lexicographic.

Calendar plus clock time is an interesting example of super-lexicographical ordering. For example, the time instant "1967 June 9 2:24:20" is a composite data element of six components. The value of the first component is an integer. The value of the second component is a member chosen from the 12-member domain of months. The value of the third is taken from a domain which is a function of the month; in this case, it is a 30-member domain of integers from 1 to 30. The value of the fourth component is a member of the 24-member domain of hours (military time). The fifth and sixth components have values from the minute and second domains which are both the ordered set of integers from 0 through 59. If the conventional hour designation were used instead, then an extra component would be needed, whose values came from the two-member domain [AM, PM], and the value of the hour component would be from the ordered domain [12,1,2.....10,11]. Note that this latter domain of hour is one which has an ordering but this is not a lexicographical ordering.

DATA ELEMENTS APPLIED

In this section we discuss how the concept of composite data elements can be used to provide some of the data structures which are already familiar, and also some of the data structures which are less familiar because they are not easy to realize in current languages.

Strings as unitary symbols vs. strings as ordered lists. Having chosen the notion of string as a primitive we are now faced with a problem: how can a string be split into component parts? We have on the one hand the notion of a string as a unit, playing the role of a single symbol, necessarily indivisible. On the other hand we at times want to consider a string as an ordered set of characters, and have the ability to scan and modify this set. It is this dual role for strings which presents the problem.

It is clear that the resolution of this problem requires two ways of interpreting strings: (1) as unitary symbols, and (2) as concatenations of characters. There are two fundamentally different approaches for coping with this. The first of these is to provide a separate set of

string transformations which operate on value-strings. The second way is to provide a transformation which converts a string into a linear composite whose components are cells containing individual characters, and the corresponding inverse transformation. These latter conversions can be accomplished by means of the string transformations plus the basic transformations which manipulate data elements.

Programs viewed as composite data structures. We can view programs as data structures. In the interpretation of programs as data, we need not concern ourselves with dynamic structure, which can be much more complex than static structure. Each statement is expressible, in current languages at least, as a linear string. These strings can be tied together in a variety of ways, limited only by the ability of the sequence controller to sequence properly through the statements. The automatic, or follow-on, sequencing between statements (which is different from the sequencing dictated by explicit "jumps") and the grouping of statements into procedures correspond to the arrangement of conventional data elements into composites. A segment of a program is then an ordered set ("list") of strings. The components can be named (can have identifiers) and the composite as a whole can have identifiers, which correspond to names of segments or procedures. Composites more complex than ordered lists may be useful: an example would be parallel ordered lists.

Program statements would normally be stored as uninterpreted strings, the interpretation being deferred until the moment of execution. It may on occasions, however, be useful to represent a parsed statement, for which a "tree" structure is fairly natural. Block structure, as in ALGOL and PL/I, is also representable in tree form.

Observe that a program can now be described as a sequencing through a composite data element whose components are strings interpretable as statements.

Text-handling. The processing of text presents a problem with regard to the utilization of composite data elements. This problem is related to the dual nature of strings discussed in Chapter 3. For the purposes of scanning, a body of text might conveniently be thought of as a single string. However, the user may wish to do such things as place markers within the string, and attach identifiers to substrings within the string. For such manipulations, the strings needs to be a composite data element

where the characters adjacent in the text are tied together by some precedence relationship. It is the user's responsibility to have the text in composite form in order to perform such manipulations. To convert between strings and composites will require some primitive transformations, since a string is properly an integral symbol not amenable to being viewed as a composite. The specific text-manipulation abilities which are desirable are discussed in Chapter 5 on Transformations.

A pointer or marker is useful to indicate a point of "current interest" in some text. Some ways of constructing pointers within the framework of composites are shown in Figure 4-11.

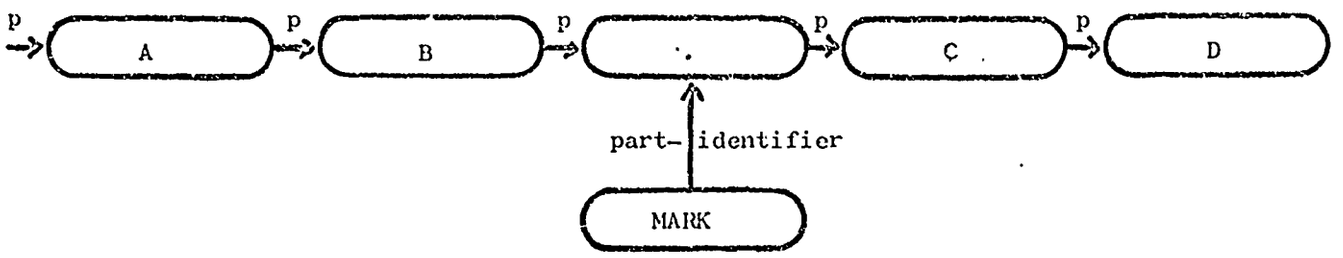
Files. A file, in the conventional sense, can be viewed as a set of elements. The set, if unordered, is a random file (corresponding to a disc file). If ordered, the set is a sequential file (corresponding to a tape file). In the abstract, however, a file is simply a set, ordered or not as we may choose.

Trees. Figures 4-12 and 4-13 graphically portray two trees, showing data elements at the nodes, with nodes related by a single type of precedence relationship. Each such relationship is expressed by a composite, represented in the figures by an arrow.

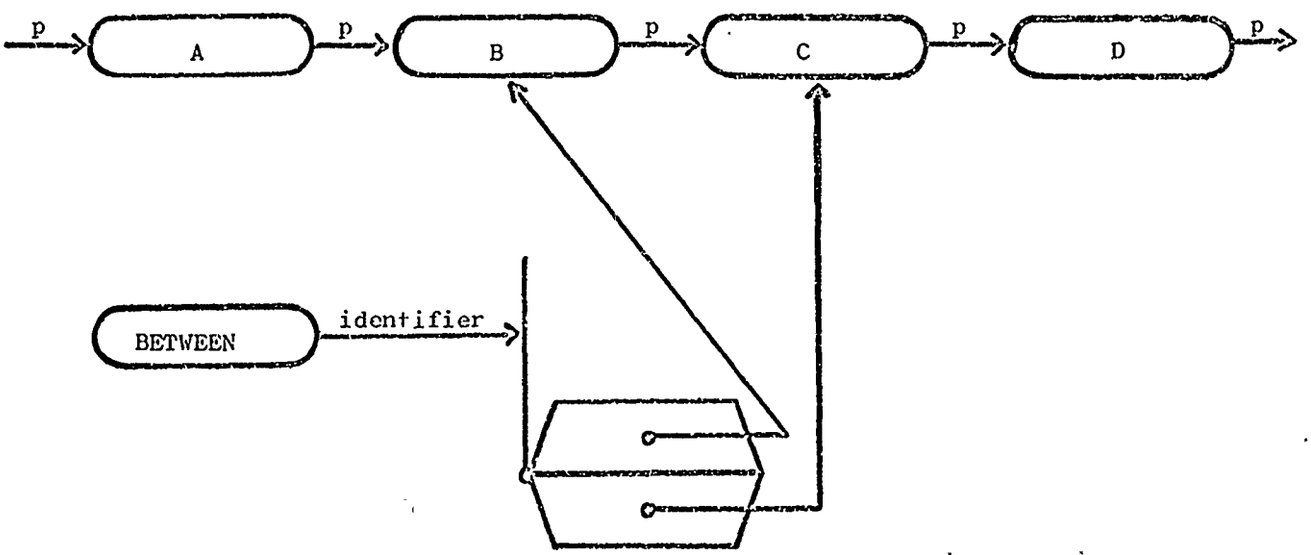
Matrices and multi-dimensional arrays. While mathematically a rectangular array can be considered a vector (ordered set) of vectors, this viewpoint is not adequate for my purposes. One of my criteria of adequacy for a theory of data elements is that it should be possible to determine easily the answer to the question: What elements are associated (related) directly to a given one? For example, we can easily obtain this answer in the case of two-dimensional arrays by defining elements to be adjacent if their row subscripts are identical and their column subscripts differ by one unit. Such an explicit adjacency should be one which will work equally well for other arrays than rectangular ones: such as triangular arrays and tetrahedral arrays.

Figures 4-14 and 4-15 graphically portray a rectangular, 2-dimensional array, and a symmetrical triangular array, respectively. The small letters stand for specific relationships. In the case of the rectangular array, the two relationships are "horizontal" and "vertical". In the triangular array, the relationship letters "a", "b", and "c" are the three coordinates of the array. The regularity of arrays obviously suggests

Insert a data element (character) which stands for the marker.



Construct a marker relationship "between"



Use movable identifier "after"

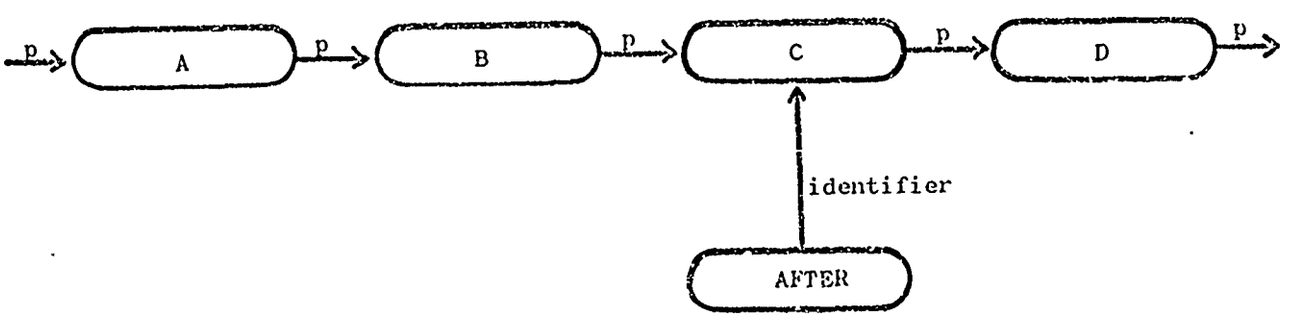


Figure 4-11. Ways of Constructing a Pointer or Marker in Composite Representing Text: Marker Pointer Between "B" and "C".

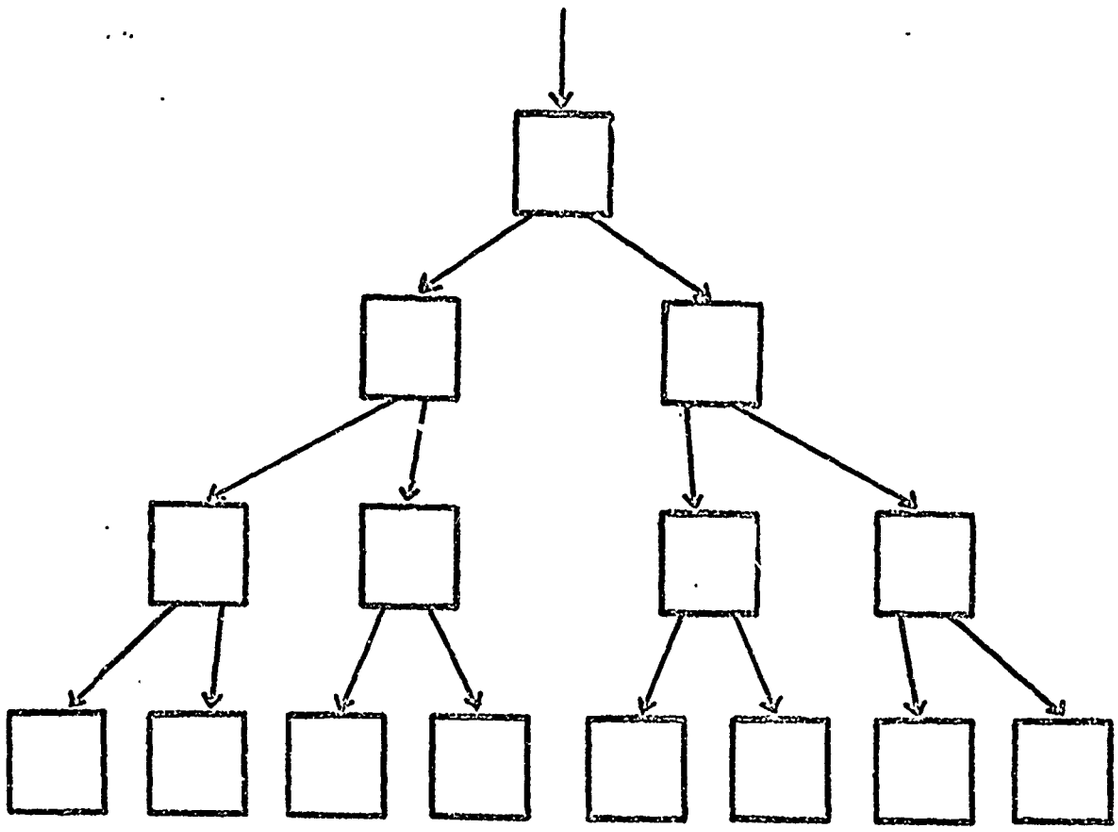


Figure 4-12. Symmetric Binary Tree

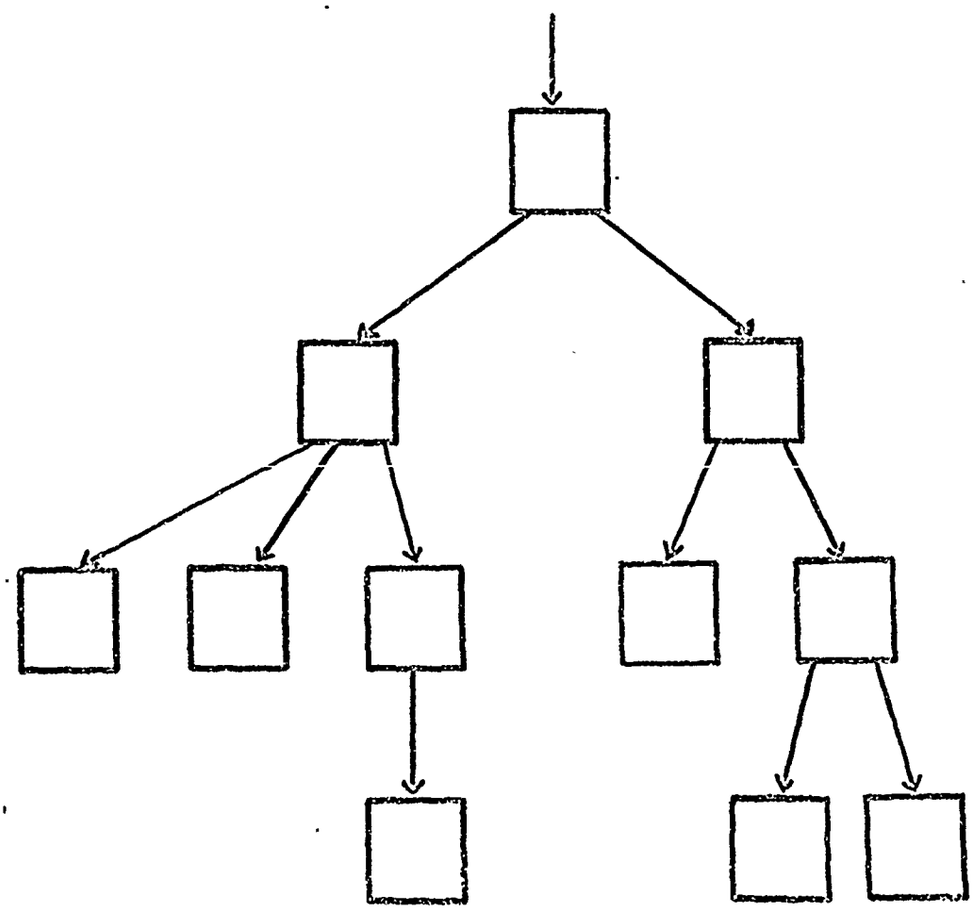


Figure 4-13. General Tree

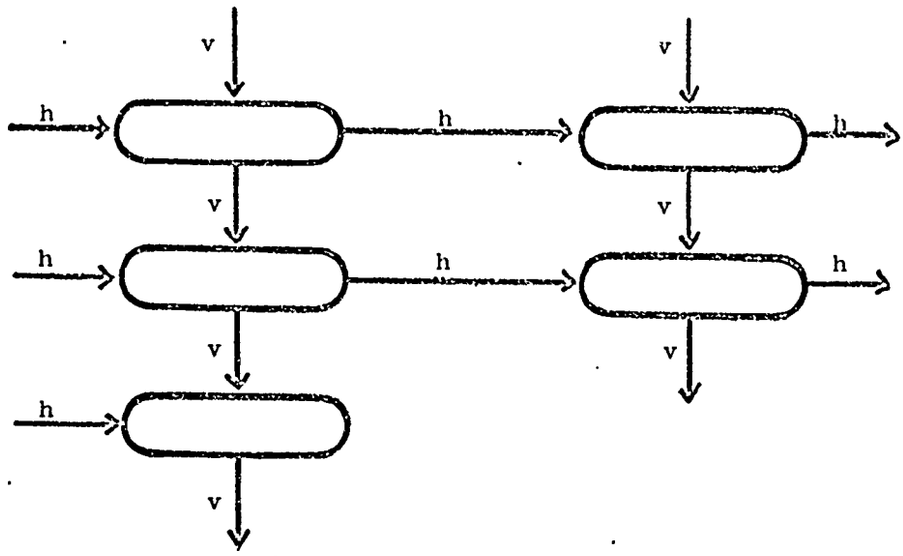


Figure 4–14. Rectangular Array Pictured as a Composite Data Element

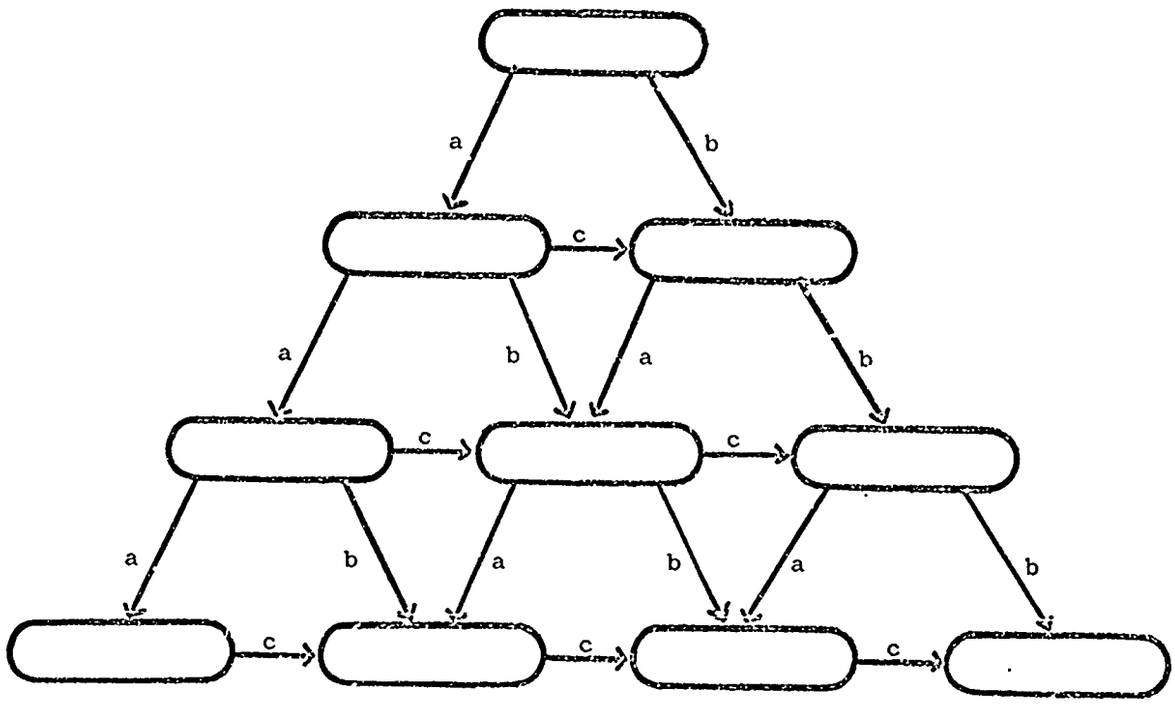


Figure 4–15. Triangular Array Pictured as a Composite Data Element

that arrays can and should be constructed by iterative or recursive processes.

Tables. The conventional table is a rectangular array of data elements. It is distinguished from the usual mathematical array or matrix by the fact that it can contain non-homogeneous elements. In general, the elements are organized into classes by rows and into other classes by columns. If another dimension is needed, the table may be further organized into classes by pages. A data element is selected by specifying a row-class and a column-class (and perhaps a page-class). We could say that the members of a row have an ambiguous identifier, as do the members of a column; these ambiguous identifiers used together refer to only 1 element in common. This view of a table then is one of sets, certain intersections of which have unique members. It may be the case that the members of a column are not independent, as in certain portions of a decision table. In such circumstances, it may be more convenient to regard a table as a set of lists, where each list corresponds to a column.

The individual elements of a table may not have any meaningful relationships between them other than the physical relationship of juxtaposition when the table is displayed in two or three dimensions. This should be intuitively obvious, since we can have a meaningful four-dimensional table but we cannot display it. The relationship that often exists is more one of similarity of element names. Yet even this does not hold in a table whose indices are non-numeric (that is, where the concept of adjacency of index values does not exist).

By extension of the notion of table, we can have a complex table, in which the elements are not constrained to be simple, but rather can be composite elements. An element might, for example, be an ordered set of elements. To access the lower-level elements would then require a two-step access. We observe that the language known as "Decision Tables" is an application of complex tables.

Another view of tables is that they are a special case of Cartesian space. The n-tuple is regarded as the ordered set of coordinates. The element at the intersection of the coordinates can be of any type; in particular they may in turn be composites in Cartesian space.

It is perhaps more helpful and more meaningful to view a table as a set of n-tuples in which the first elements are from a single domain, the second elements are from a single domain, etc. Another way to say

this is that the n-tuples "have a model". A row is selected, or a set of rows are selected, from a table by specifying certain "match criteria" for specified positions ("columns" of the table). Results, or "output", is selected by naming the desired parts (positions). In a table of n-tuples in a specified order (that is, the table is "an ordered set of n-tuples"), a row can be selected by giving the table name and an ordinal argument or subscript. However, if the table takes the form of an ordered set of elements within an ordered set, we have a structure isomorphic to a tree. As we note elsewhere under tree-naming, access to a terminal node ("leaf") of a tree can be done by giving the tree name followed by an ordered set of subscripts. The ordered set of subscripts performs the selections at successively lower levels.

Note that a function table is a special case of a table, in which, for a given set of input arguments, only a single n-tuple is selected (which is another way of saying that the result of a function must be unique). Note that the defining table itself need not contain unique results, but the lookup algorithm must yield a unique result, if only by such a simple device as not looking further once one result has been found.

The multilist and multiset concepts. The multilist concept, first suggested by Prywes (1962), provides for the appearance of data elements on more than one list simultaneously. Example: Consider the set of elements A,B,C,D, and E. Some or all of these elements could appear on several lists. Composite #1 could be a list of the elements (or, more properly, the sequence of element names) "A", "C", "E", "B", "D". Composite #2 could be simply the one-element list "D".

The sequence of elements in a composite may not matter, however. In such a case, we would have sets instead of lists, and the concept might analogously be called the "multiset" concept. It provides for the simultaneous appearance of data elements in more than one set. Continuing with the example above: Composite #1 could be unordered set of element names "A", "C", "E", "B", "D"; composite #2, the set "B", "E", "A", composite #3, the set of one element "D".

My method of expressing composites provides equally well for both the multiset and multilist concepts. Note that there is no limit on the complexity of the data elements which may be components of the multilist or multiset.

Data elements of highly variable structure. There are types of data elements which not only are not fixed in configuration, but also which

have structures which change on nearly every access. Examples are: queues and pushdowns. Their common characteristic is that the structural framework is in a sense fixed though unbounded. That is, we know in advance and by convention how each new element is to be added to an existing structure, and how each old element is to be removed from an existing structure. The variable data elements employed to date have been largely of the homogeneous type: the component element types being identical and the relationships being identical, as in LIFO queues (pushdowns) and FIFO queues. However, there is no reason why we should be limited to structures having identical elements and identical relationships. We could have an iterative structure, for example, in which a fixed substructure occurs repeatedly. Where the variation in a data element is one of variable repetition, a model is sufficient to express this. When the variation is more complex, however, a better solution is to use transformations which produce the appropriate alteration in the structure when such a transformation is invoked.

CHAPTER 5. CONCEPTS OF TRANSFORMATIONS

GENERAL REMARKS

Definitions. A "transformation" is an action, a program step, which causes a change in storage. This change of storage may not be the storage which the user "sees", however; it may be inside the processor, and changed in a way which is defined for user only behavioristically. Declarations (in the ALGOL and PL/I sense) and definitions of various types are therefore viewed and described as transformations. Later the various kinds of transformations will be distinguished.

A transformation is defined by a sequence of transformations, commonly called a "procedure". The ultimate definitions are in terms of transformations which are primitive (not defined within the user language, the language interpreted by the processor).

A transformation is invoked, or activated, by a "transformation call". The form of a transformation call is the familiar function notation of a transformation name, possibly followed by a list of parameters. Examples: T; T(A,B,C);.

Sources of transformation definitions. The stock of transformations which the user has at his disposal must be defined. There are several ways in which a given transformation definition becomes available for a programmer's use. Some fixed set of transformations is provided as primitive. These are the fundamental building blocks out of which the user must build everything he needs.

A second set of transformations is provided as a "basic set". These are defined in terms of the primitives and are modifiable by the user. These are the frequently-used transformations which are provided merely as a convenience to the user. The user is free to select those he needs, modify some, and discard the rest.

The third source of transformations available to the user is that set which he defines for himself. How much work he has to do to define new transformations depends on what has been provided for him in the primitives and in the basic set, and on the amount of freedom he has in defining new transformations.

A given programming language is to a large extent characterized by the amount of freedom the programmer has in defining new transformations,

and the convenience with which he can do it.

Functions vs. transformations. Among the parameters of transformation calls may occur explicit function calls. The format of a function call may be identical to that of a transformation call, namely: a function name followed by a list of parameters (arguments) in parentheses. Since function calls and transformation calls can have identical format, this leads naturally to ask what the essential difference is between a function and a transformation. A function is simply a transformation that has a single "result". A result may be an identifier of a value-string, or it may be an identifier of some arbitrarily-complicated data element.

Some languages make the distinction that functions do not permanently modify storage; that is, they can change only local variables. Another way to say this is: functions are not allowed to have "side effects". We do not so restrict them; a function can be written to affect an arbitrarily-chosen transformation of storage. Thus we can write functions as the parameters (arguments) of any function. (In principle, we can also write as parameters transformations which are not functions. As they will each return a null value, it would seem pointless to write such transformations as parameters.)

Some languages, such as LISP, require that all transformations be functions. This means that all non-function transformations must be treated as functions producing null values. This is workable but somewhat clumsy.

Defining an arbitrary transformation as a function requires some care in the definition to avoid ambiguity. Consider the list of lists:

((a,b), (c,d), (e,f))

Suppose we want to perform the transformation of interchanging c and d. What do we consider the "value" of the result? How much of what we started with is the result? Is the result "(d,c)"? Is the result "((a,b), (d,c), (e,f))"? The definition of the function must tell whether, after execution of the function, the original data structure exists? Is the result of this function a modified original or a modified copy? We can of course define it to be either choice. LISP demands that it be a modified copy, which is to my mind not flexible enough. We should have

the flexibility of defining it either way.

FUNCTIONS

Representation of functions. What is a function, in relation to a computer algorithm? There are two fundamentally different forms which functions can take; alternatively put, a function can be represented in two different ways:

1. by enumeration, a set of ordered pairs; the first member of each pair is a set of arguments, and the second member is the corresponding result (which may be composite).
2. by algorithm, a computational process which maps the arguments into a result.

Notice that the algorithmic representation of a function depends on the fact that a concept is represented in the form of a value-string and that the components of this value-string can be individually inspected. Thus a function defined on value-strings is defined in terms of other functions on the components of these value-strings. These defining functions may in turn be defined in a similar fashion. The ultimate definitions, however, must be either primitives, or functions expressed by the technique of enumeration. For example, the addition function for decimal integers is reducible to an algorithm based on the addition table for decimal digits, where the table is an enumeration.

Both representations of functions can lead to an identical result. In a given circumstance, one technique will generally be strongly preferred over the other. An ill-behaved function, for example, is likely to be difficult to compute but relatively easy to represent by a set of ordered pairs (a function table). Where standard interpolation techniques can be used, such function tables can often be shortened. Where there can be a large number of sets of imput argument values, however, a computation (procedure) will usually be preferable to a function table. Where either representation of a specific function may be satisfactory from the standpoint of readily producing the proper result, the choice of representation may be dictated by the type of modification that the function is to undergo.

Where a function is represented as a procedure, it is structured and treated as a transformation, a set of imperatives. Where a function is represented as a set of ordered pairs, it is a composite data

element.

Note that a function table is a special case of a "table," which can be defined to be a set of n-tuples. A table which is a function is distinguished by the fact that for a given set of arguments, only a single n-tuple of the table is selected; this is another way of saying that the result is unique. For an ordinary table there is no requirement that the result be unique.

Because we allow this dual representation of functions, we must be prepared to have function calls carried out properly regardless of which of two modes of function representation is used, and preferably without the user having to distinguish between the two modes in the way he writes the call. To accomplish the latter, that is, to make the calls of the two types of representation indistinguishable, both types must have a command interpretation. Functions of the algorithmic type naturally have a command interpretation. Functions of the enumeration type are expressed as composite data elements. To give these a command interpretation, we must apply some accessing function to the data element.

To illustrate the preceding discussion, consider the function expressed by the following table:

	A	B	C
A	B	C	A
B	C	A	B
C	A	B	C

An algorithm f which would realize this function is:

```
f: result = if ⟨1⟩ is 'C' then ⟨2⟩ else
            if ⟨2⟩ is 'C' then ⟨1⟩ else
            if ⟨1⟩ is 'B' & ⟨2⟩ is 'B' then 'A' else
            if ⟨1⟩ is 'A' & ⟨2⟩ is 'A' then 'B' else 'C';
```

In table form this function would be written

⟨1⟩	⟨2⟩	Result
A	A	B
A	B	C
A	C	A
B	A	C
B	B	A
B	C	B
C	A	A
C	B	B
C	C	C

To make this table representation a callable function, we need a table search function to operate on it. In order to design a search function we must have in mind a specific data structure. The data structure of the table need not conform to some uniform standard. There is a variety of data structures and companion searching functions which would serve.

To summarize, a user-defined function will be some procedure called by means of a standard procedure call. There are two philosophically different forms of function procedures; each form has its advantages and disadvantages. One form is an algorithm which computes a result according to some set of rules; the other is some table lookup procedure which finds the result in a table.

Predicates. One of the most common types of function used in programming is a predicate, a function which yields a result of either "true" or "non-true". An example of the use of this function is as the Boolean expression part of an IF-THEN statement. An important use of this type of function will be the determination of the existence of specified data elements and relationships. Such functions will of necessity be defined by rather complex algorithms which search and inspect composite data elements.

Nature of the result of a function. From a mathematical standpoint, the result of a function is an unnamed, unstored value. In contrast to this, the result of a function from a data processing point of view is an entity which is both stored (if only in some temporary location) and named (otherwise it could never be accessed for a subsequent processing step). In the data processing case, a result could be a value-string (if it is a simple data element). If the result is a composite, however, it cannot have a value-string as a result; in such a case, the only meaningful result is the identifier of the resulting composite.

With this discussion as background we are ready to consider the issue of what is an exit-value of a function. That is, what is the meaning of a function call when it is written as a parameter? If the function yields a value-string, its exit-value could be that value-string. If the function yields a composite, which by definition has no value-string, then the exit-value could only be the identifier of the composite. Since we have already established that every value-string which is the result of some function has an identifier (supplied automatically by the processor, if necessary), we take the simpler point of view that the exit-value

of every function call is an identifier of the result.

Result domains. I earlier expressed the philosophy that it should be possible for the user to prevent the computation of "garbage"; that is, to be able to write programs such that invalid values of data elements cannot masquerade as valid ones. We should accordingly think of a result as an ordered pair, consisting of a type designator (usually a domain designator) and an associated value-string or composite. In order to implement this philosophy, it is necessary to provide explicitly with the result of a function calculation the "type" of the result: that is, its domain, or class. Exactly how this type information is to be expressed and stored has been treated in detail in Chapter 4.

Storage allocation of function results. A problem exists in connection with storage allocation of results of executing functions. If each function leaves its result in a standard place, say in "RESULT", then some conflicts of storage can occur when asynchronous parallel operations are allowed. Even if each function or each execution leaves its result in a location unique to that function, the same kind of conflict can occur. This conflict can occur both in the case of asynchronous parallel execution of the same function, and in multiple occurrences of the same function in a single parameter list. The completely general solution, and the one which I favor, is to generate a new (previously unassigned and unused) storage location for each result. The difficulty here, of course, is that the available storage gets used up if there is no automatic method for reclaiming it. Such an automatic method might be easy to specify—namely that the result can be used only once and then the storage cells it occupies are to be made available for new assignment. (This applies only to the result, not to side effects.) Another possible resolution of the problem is to provide a name for a function result as part of each call—then it is up to the user to avoid storage conflicts; however, even this technique may fail when asynchronous parallel processing occurs!

Choice of domain of a transformation result. We have considerable flexibility in how the domain of a result of a function is to be specified:

1. The simplest case is to have one fixed type, specified by the writer of the transformation definition. Example: A concatenation function could be defined on two parameters of type string, yielding a result of type string.

2. The type of the result could be defined to be "appropriate" to the types of the input parameters. Example: a sum function could be defined on two parameters of the same type (that is, both integer, or both real, or both complex, or both vectors of integers, etc.), and the result could be defined to be of the same type as the input parameters (arguments).
3. The type of the result could be specified by an input parameter. Example: We could define a procedure which computes area, given length and width parameters in some linear measure such as inches, feet, yards, etc. One of the parameters of this procedure could specify the domain of the resulting area, that is, specify that the result is to be given in square inches, square feet, square yards, etc.

Methods 2 and 3 above require the existence of program statements which can determine the domains of the input parameters.

Extension of functions. We would like to have the analog of the mathematician's ability to extend a function to deal with a different algebraic structure. Example: let the function SUM (A,B) be defined, say, for integers. We would like it to be extended to treat complex integers, then to be extended again to treat rectangular arrays of complex rationals. It should be apparent from the preceding discussion that such an achievement is not difficult. The procedure which defines the function SUM must analyze the types of the arguments and select a computation sequence appropriate to the argument types; naturally the appropriate computation sequence must also be added where needed.

CHARACTERISTICS OF TRANSFORMATIONS

General types of transformations. It is convenient for the purposes of study and discussion to organize the types of transformations into three general categories: transformations of data, of context, and of sequence control. Each of these will be discussed in turn.

"Transformations of data" are those processes which change data elements, or compute functions, or both. Changing a data element includes creating and modifying relationships between storage cells. Functions have been discussed earlier; the reader is reminded that the computation of a function may or may not permanently modify data elements.

"Transformations of sequence control" include all those transformations which modify special storage cells uniquely associated with the "control element" of the processor. Also included are iteration statements which control repeated execution of a set of statements. Further discussion of this type of transformation must await a treatment of sequence control later in this chapter.

Transformations which are neither of the above types are classified as "transformations of context", which can be further subdivided into 3 subclasses:

1. Transformations of domain can define a new domain of members, add members to a domain, delete members from a domain, define a subdomain, set up equivalences between members of domains (which can be done either by correspondences or by algorithms), define orderings on members of domains. Members of domains need not occupy storage cells; they may instead be defined by algorithms. Thus transformations of domain may involve the modification of algorithms; since in these developments we regard algorithms as data until the moment of interpretation by control, these modifications can be carried out as modifications of data.
2. Transformations of processor action concern modes of processor action such as types of evaluation or calling. Examples of modes of processor action not present in some languages are "macroexpand" and "evaluate without becoming undefined."
3. Transformations of specification can set conventions to be used whenever specifications are deficient or non-contradictory. This corresponds to the "default attributes" of PL/I.

Recursive calls, recursive procedures, and reentrant procedures. Notice that a "call" to a transformation can be interpreted in several ways:

1. Remember the return point, move the program control pointer to the entry point of the transformation, and proceed with normal execution.
2. Remember the return point. Make a fresh copy of the transformation (i.e., the data structure which will be interpreted as program). Move the program control pointer to the entry point of the copy and proceed with normal execution.

The occasions when this distinction matters are in the case of reentrant or recursive procedures or where a procedure modifies itself. On these occasions it indeed matters whether one has the effect of copying or not. What actually happens in the processor is not of consequence here. What does matter is the behavior of the processor as seen by the user. The presence or absence of the copying effect strongly influences how the procedure must be written.

Notice that a recursive call (a call to transformation T as a parameter of another call to transformation T) can be considered a special case of recursion, since the procedure in effect calls itself not during execution of the procedure, but during the "prologue" to the procedure during which its parameters are being processed.

The most usual way of handling recursive procedures is to have the rule that only the "latest incarnation" of the procedure and its associated variables are accessible. This requires that all exits from those procedures must be orderly so that everything can be carefully unstacked. This rule implies, although it does not require, that no copies are made of the procedure. This limitation of access only to the latest incarnation makes it difficult to refer to data in previous incarnations, since they must necessarily be passed as parameters in order to be referenced.

In these cases involving copying, we necessarily have multiple uses of identifiers, and hence have ambiguities of reference. These must be resolved in some systematic way. One possible solution is to have the "local" variables automatically converted to lists with incarnation numbers automatically supplied as indices. In turn this raises the problem of defining scope of variables, an issue heretofore avoided.

The point of all this discussion is to raise the issue of what a call to a transformation really means, and demand that the issue be clarified. Does copying of the transformation occur? If so, under what circumstances? Can the user control when copying occurs, or is it determined by the system designer? If copying occurs in the cases of recursion, how are ambiguities resolved?

It is obvious that in the case of a procedure having formal parameters, the prototype of this procedure should remain untouched, so that it can be reused to satisfy additional calls of the procedure. Hence some copying of the prototype should take place at each call. Exactly what

should be copied is the matter for consideration.

I digress briefly for a comment. There has been much interest in recent years in "pure procedures", procedures which do not modify themselves. It has been argued that pure procedures have two advantages; that they are easier to debug, and that in a parallel processing environment they can be used simultaneously and asynchronously by multiple threads of control (can be used "reentrantly").

The argument about "debugging" is perhaps true, in the sense that some users find it less confusing if procedure bodies are not modified. However, to avoid modification of a procedure body, one may have to build a slightly more complicated program. In such a case it may not be true that debugging is thereby made easier.

The argument that procedures must not modify themselves in order to be usable in a parallel processing environment stems from the storage-conserving practice of using only one edition or copy of a procedure body to serve multiple purposes. From a logical viewpoint, however, a procedure is intended to be a prototype, and each invocation of this prototype may be with different parameters. Each invocation involves a fresh copy of the prototype with appropriate substitutions for the parameters. If the invocation of procedures is in fact implemented in this way, then there is no need to restrict procedures from modifying themselves.

REFERENCING A DATA ELEMENT

Definition. A "reference" is effectively a pointer to some data element, either simple or composite. It may be convenient to think of a reference as a machine address. In a higher-level language, however, the concept of machine address is not defined; hence we might take an alternative view, namely, that a pointer is invisible and is considered to be undefined in the language. This concept of invisible pointer could be dispensed with if the processor were such that it generated an identifier automatically for every data element when an identifier was needed. I take the view that every data element has an identifier, and that every reference is a function which yields an identifier of the data element being referenced.

Expressing a reference. A reference is expressed as a "reference expression," which will appear as a parameter of some transformation. A reference expression can take any of a variety of forms. These forms are enumerated briefly here and explained later in this section. The

ways that references can be expressed are:

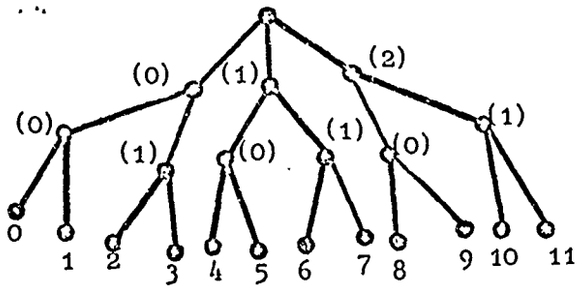
1. As an identifier. A variation of this is to have indirect referencing, in which a value-string corresponding to an identifier is taken; then this value-string is re-interpreted as an identifier. This process can be repeated as many steps as desired.
2. As a "name", which requires the existence of a name-to-element mapping function. Examples of such names are array element names and tree element names.
3. In terms of standing in a unique relationship to some other data element already "known". Examples of such relationships are: "preceding", "succeeding", "corresponding", having a specified ordinal position in some ordered composite. It is possible for the reference to be a chain of such relations.
4. In terms of satisfying some predicate, such as "is a substructure of", or "is an instance of".
5. As an identifier of a composite data element plus a part-identifier.
6. In terms of an algorithm which yields a reference; for example, an algorithm which performs a search or traces a path. A specific example is the hierarchical name in COBOL, which has an implied search algorithm.
7. In terms of a call to a function which yields a value-string.
8. By pointing, as with a light pen or cursor.
9. By exhibiting a value-string (unnamed) in quotes.
10. In terms of an intersection of classes having ambiguous identifiers.
11. In terms of some pronoun which refers by convention to some previously-determined data element.

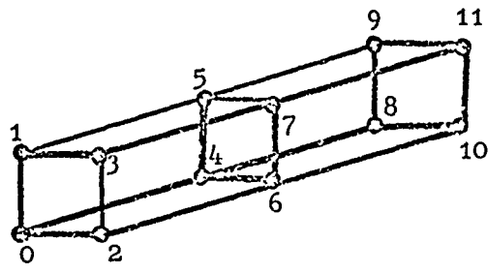
A comment on lookup versus search. I digress briefly to make an observation concerning when referencing by identifier is appropriate and when it is not. Consider a structure which allows representation of full text of, say, a book. We have, in the main text, the book as a whole, chapters, sections, paragraphs, sentences, words and characters. If we had a naming scheme down to the level of sentences, for example, we could obtain any sentence by direct lookup, assuming that there is some name-to-storage-location mapping mechanism. To go to any level of finer detail,

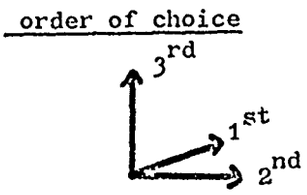
that is, to a word or character, we must resort to a search procedure. We can make the observation that in general we can expect to access by direct lookup any substructure that is named (that is, has an identifier), but accessing any unnamed substructure requires a search.

Naming components of a composite by "mappable names". We note that among the methods of naming individual components of a composite, there are the methods exemplified by arrays and trees. We observe that subscripted identifiers are a naming scheme for elements organized in an array or tree. Consider, for example, the identifier "Y[A,B,C]". This designates some composite element called "Y". The ordered set "A,B,C" tells us how, starting from some standard starting point, to proceed to a single selection. Each subscript value in turn represents a choice. Imagine a tree named "Y". Beginning at its root, we choose the Ath branch, and proceed down this branch to the next node. At this node, we choose the Bth branch and proceed to the next node. At this node we choose the Cth branch and proceed to the next node. This final node has a corresponding value-string, which is the value we have selected. If the tree is a regular one, by which we mean that each node at a given level has the same number of branches, then it can as readily be visualized as an array. Consider the Figure 5-1. This shows a regular tree and the corresponding multidimensional array. If we make one minor change, so that the tree is no longer regular, then there is no longer a corresponding array. We can, of course, add dummy branches to the tree so as to make it regular again; and if we do so, then there is again a corresponding array.

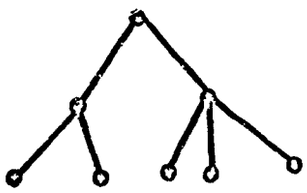
Putting what we have just discussed into more abstract language: naming schemes for components of trees and arrays simplify the naming of component elements of a composite element. The technique is based on taking advantage of regularity so as to be able to make a simple naming rule. Rather than give each element of a composite its individual name, the composite name is like a generic name, and the elements are ordered in such a way that the sequence of subscripts tells us how to trace through this ordered structure to obtain a single component element. Particularly in the case of arrays we have been conditioned to think of the subscripts as numeric ones, but there is no logical reason why the values of subscripts need be restricted to numbers.



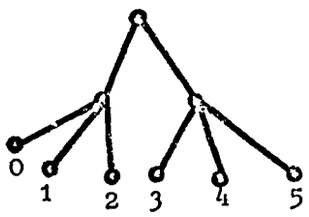


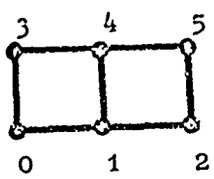


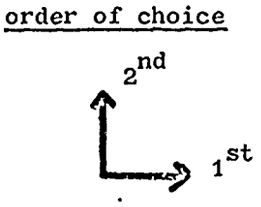
A. A regular tree has a corresponding array.



B. A non-regular tree does not have a corresponding array.







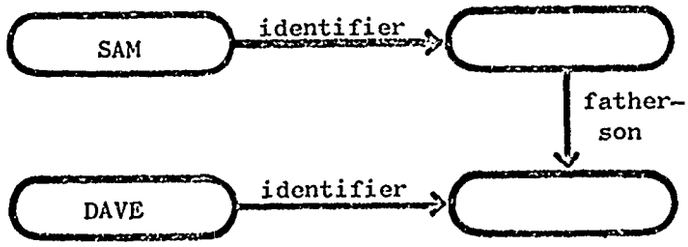
C. A non-regular tree can be padded out to make it regular and thus have a corresponding array.

Figure 5-1. Naming of Elements Via Trees and Arrays.

An example of a naming tree is shown in Figure 5-2, where a constant tree, whose nodes are company names, is accessed by the generic name "COMPANY NAME" plus a list of subscripts signifying state, city, and utility. We note that the class of naming trees that have just been described is a special case of the class of trees having arbitrary relationships between the nodes, which is in turn a special case of networks with arbitrary relationships between nodes.

Little has been said about the name-to-element mapping rules. It should be obvious that there must be a way to indicate when a name-to-element mapping rule is to be invoked, and which such rule is to be applied.

A general data referencing function. The identifier function is in turn a special case of a more general data referencing function which we might call Rel. The arguments of Rel are the identifier of a given element and a relationship of the given element to the desired element. For instance, if SAM and DAVE are identifiers of two data elements, pictured thus



and "father-son" is the relationship of the first element to the second, then

Rel (SAM, father-son)

would be synonymous with DAVE. That is, it would be equivalent to DAVE for the purpose of accessing the same data element. Note that whether DAVE has a value or not is irrelevant here: DAVE might be a composite data element and not have a proper value.

Ambiguous relationships. The system being developed here allows arbitrary relationships between arbitrarily-chosen elements. In such systems, there is the possibility of more than one element being in a given relationship to a given element. I call such a situation an "ambiguous relationship". The user must be conscious of whether such ambiguous relationships actually occur in his data. The succeeding paragraphs explain why.

Consider first the case where a, b, and c each stand in the relationship R_1 to x. Diagrammatically:

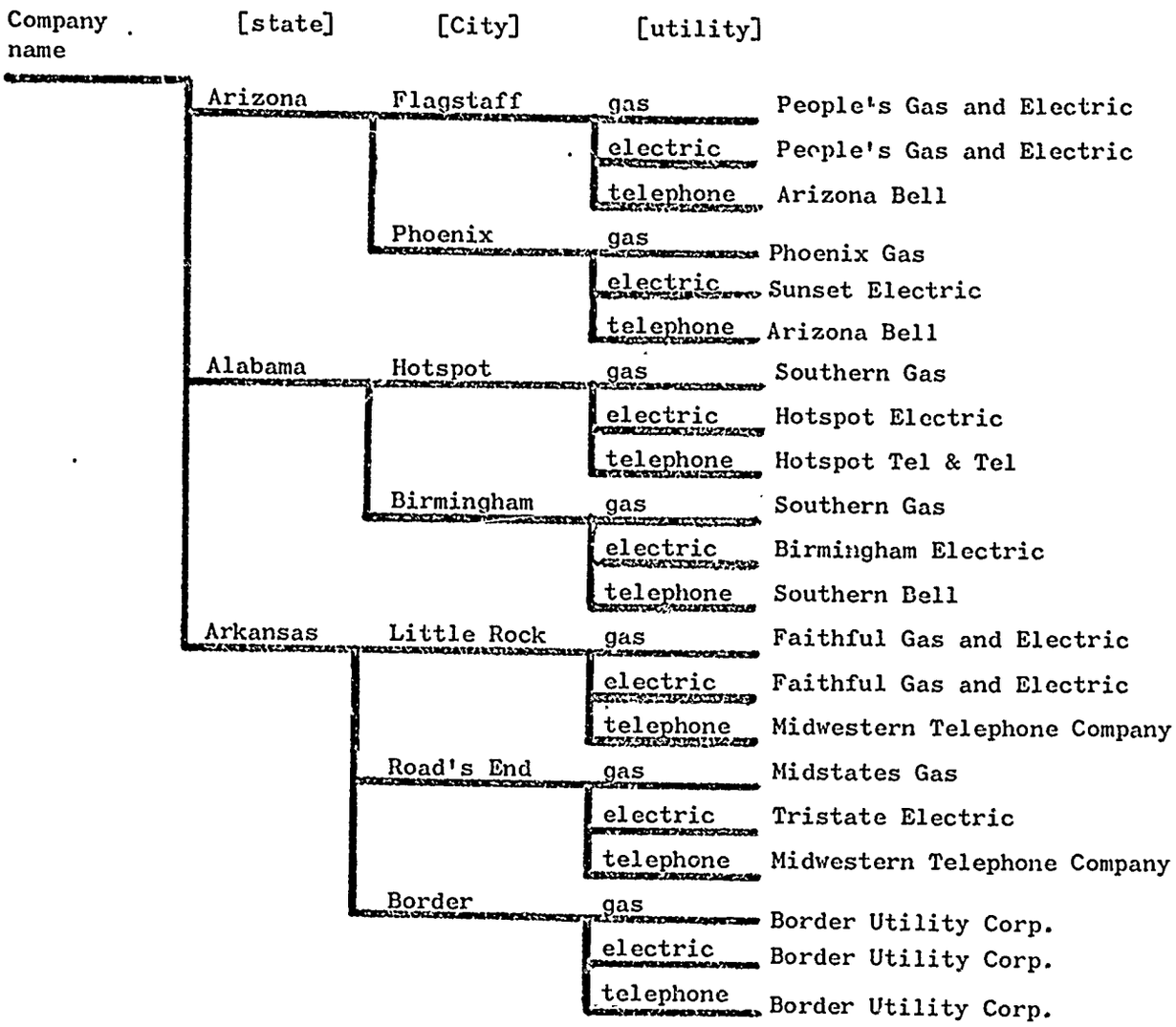
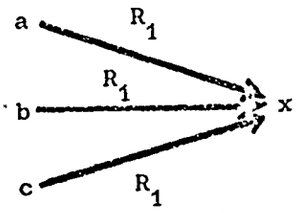
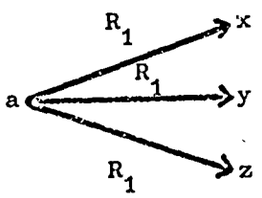


FIGURE 5–2. Example of a tree of names



To make the illustration more concrete, think of a, b, and c as identifiers of the object x. Suppose now that the object x has been referenced by some means other than by any of its identifiers, and that we wish to be able to determine one of its identifiers. We must be careful what we ask. If we say "What is the identifier of x?" , which is more precisely stated "What is the unique identifier of x?", the question has no proper answer. To give as the answer "The set a,b,c" is perhaps technically correct but not very useful. To get a proper answer we must instead ask "What is an identifier of x?", which itself is an abbreviated way of saying "What entity stands in an identifier relationship to x?" The user must in general know, or have a means of finding out, if this question has a unique answer. If it does not, he might ask the same question twice in his program, and get a different answer each time; this may lead to a program error.

Consider a second case, where a stands in the relationship R_1 to x, y, and z. Diagrammatically:



To make the illustration more concrete, think of x, y, and z as elements each individually identified by a. Suppose now that we wish to make a reference to y. Clearly a does not uniquely identify y. We must give other relationships concerning y, either instead of, or in addition to, a. Suppose we say "the element identified by a". This is meaningless. By

a slight stretch of the imagination we could interpret it to mean "the set x,y,z;" to treat x,y,z as a set when it has not been explicitly recognized as such may lead us into trouble. (It will have multiple components where only one might be expected.) We might say "The set of elements identified by a." This is meaningful and correct. It is important to recognize that it is different from the two substantives just given above. It is also meaningful to say "An element identified by a." As in the first case, the user must in general know, or have a means of finding out, if this substantive refers to a unique object. If it does not, the programmer may use it twice in his program, and refer to a different object each time, thereby leading to a program error.

In summary then: if more than one element can stand in a given relationship to another element, the user may have to take precautions to avoid nonsense and ambiguity.

Referencing a value-string. There are two fundamental ways of referencing a value-string:

1. exhibiting the value-string within paired quotation marks.
2. calling a function, which yields an identifier of a value-string as a result. Where a domain has an ordering imposed on it, there is another means of referencing a value-string: that of giving the domain name and the ordinal position of the desired element within that domain.

A single representation may name several members in different domains. The letter "T", for example, is the name of a letter of the alphabet, and is frequently used as a synonym of "true", a member of the domain of Boolean values. In cases where a representation is ambiguous, the ambiguity must be resolved by an accompanying domain designator. The domain designator tells in which domain lies the member that the representation is intended to represent.

Pronouns. It would be desirable to have expressions which roughly correspond in function to pronouns in English. Specifically, we might find the following concepts useful:

1. The last (or next-to-last) element accessed.
2. The result of the most recently executed transformation.
3. The name (or a name, or the principal name) of the statement most recently executed.

4. The name of the next return point, established at the last temporary change of control.

What pronouns can be made available depends on the nature of the data accessing mechanism, and what pronouns actually are available depends on what the designer and the user together have built into the data accessing mechanism.

PROCEDURES AND PARAMETERS

Abstractions from programs. It may be the case that a program contains some similar pieces. It may be convenient, either to save the programmer's time, to simplify the program, or to conserve machine storage, to abstract these pieces from the program as it might have been originally written. By "abstract" we mean to remove them from the main body of the program and to add to the program something (an "abstraction") which represents all of these pieces. Such an abstraction is often called a "closed subroutine" or "procedure". Wherever the abstraction is not "perfect"—where two pieces abstracted from a program do not match—the differing elements are not abstracted. In their place in the abstracted procedure are put placeholders, usually called "formal parameters". These formal parameters are replaced at execution time by "actual parameters" which are supplied in the "call".

Thus we see that any two or more pieces of a program can be abstracted and replaced by references (calls) to a common procedure. The greater the number of differences among the pieces, the greater the number of parameters that must be used. A major object in writing a program is to abstract as much as possible without making the number of parameters excessive. The tradeoff is always a matter of individual taste and judgment.

Calls and parameters. At the point in a program where a piece of program has been removed ("abstracted") we put in its place a call—a reference, by identifier, to the procedure which was abstracted—followed by any parameters which may be needed. In some languages, some calls are distinguished by being preceded by the word "CALL". This explicit way of marking calls has some advantages which will become apparent later.

There are two main places, from the grammatical point of view, that calls can occur. A call may occur in place of a command (or "imperative", or "sentence", or "statement", or in place of a sequence of statements, sometimes called a "compound statement"). Then we say the call is a "procedure call". A call may occur as a substantive (or "operand", or "parameter"), and it is therefore expected to have a "value"; then we say it

is a "function call".

The distinction in actual use between calls to procedures and to function procedures is convenient for purposes of classification and explanation, but it is not a necessary one. It is allowable in some languages to have a call to a procedure which call is used as an operand. Since in such a case some value is expected, the resulting value is conventionally taken to be null. Conversely, a call to a function procedure can sometimes be written where a statement is expected. In such a case, the "value" of the function procedure is discarded; the function procedure is being used only for whatever "side effects" it may have (that is, for whatever changes in storage that it makes).

Call execution. What happens at the time a procedure call is executed is the following:

1. A copy is made of the called procedure.
2. Each actual parameter, if there are any, may be "treated" in any of several ways. The choice of treatment method can be based on the form of the parameter or it can be specified in the body of the procedure. The kinds of treatment methods which have been popular are discussed below.
3. Each treated parameter is substituted for a corresponding formal parameter in the body of the procedure. How the correspondence may be indicated will be discussed later.
4. The procedure is executed.
5. If the call plays the role of an operand (that is, if a value is expected), the result of a function procedure, or a "null", is substituted in place of the function call.

There is a variety of requirements that I would like to place on the procedure call mechanism, all contributing to flexibility. Some of the general requirements imposed earlier are:

1. It should accommodate recursion. That is, a given procedure must be able to call itself ("recursive body"), and an argument of a call to a procedure may be a call to the same procedure ("recursive call").
2. A procedure must be able to modify itself.

Additional requirements are:

1. The processing of the parameters of a call should be under complete control of the procedure. That is, the procedure may decide,

during execution, which parameters are to be picked up from the call, and how they are to be processed and substituted. The expression "how they are to be processed" refers to whether a parameter is to be interpreted or simply to be left uninterpreted, and if it is to be interpreted, how it is to be interpreted.

2. The parameter passing mechanism should work properly even in a parallel asynchronous processing environment.
3. The parameters in a call should be able to be matched with the formal parameters either on the basis of position in the call ("positional parameter") or on a partial match of the actual parameter ("keyword parameter").
4. A procedure should be able to access the call of any specified "incarnation" of any specified active procedure, not just a call to itself and not just the most recent call.
5. Two types of substitution techniques are needed for substitution within the body of a procedure:
   a. To be able to replace occurrences of an arbitrary substring by an actual parameter.
   b. To be able to replace elements on the basis of giving their identifiers, or to replace substrings of such elements.
6. It should be possible to have more than one standard procedure call mechanism.

The matter of grammar. To my mind, grammar is an issue almost wholly separable from other programming language considerations. A set of grammar rules and a grammatical processor is necessary for what? For finding out what a given statement "means". In the programming language context it can only mean one thing: a call to a transformation. A call to a transformation has a standard format: a transformation name possibly followed by a list of parameters. If all the transformations are written in this standard format, there is exactly one grammar rule. This reduces the issue of grammatical analysis to a triviality.

Now it is of course true that programmers like to express transformations in other formats. For example, the normal algebraic expression with its infix notation for functions of two arguments is a popular one. A programming language usually has a variety of statement forms. For example, in ALGOL, some of the basic ones are: assignment, go to, conditional, iteration, procedure definition, declaration.

One of the virtues of a compiler over an interpreter is that, with the former, grammatical analysis of a given statement has to be done only once. However, notice that if the programming language consisted of only one statement form, this advantage would disappear. We note also that many current compilers are not pure compilers. Much of what they do is in fact interpretive: such things as dynamic declarations, execution of format statements containing variable specifications, etc. Hence I conclude that the advantages of real compilers over interpreters is not as great as many would like to believe.

The control character interpretation problem. A classic problem in interpreting strings of characters is to distinguish those characters which are "controls" from those characters which are "objects" (not to be further interpreted at the moment of scanning). Examples of control characters are: quote marks, paired brackets or parentheses of various shapes, commas, and blanks. Problems of interpretation arise when a character normally used as a control character is desired as an object character. How do we say unambiguously that a certain character is not to be given its normal control interpretation? The answer is that we must establish some workable convention. There are several workable conventions that will serve except in rare pathological cases. (No matter how elaborate the convention, I think one can always construct a case in which the convention won't work, so we must be content with a reasonable convention that works except in rare cases; these latter will have to be handled individually on an ad hoc basis.)

Convention 1. Any intentionally unmatched bracketing character or quote mark shall be immediately surrounded by quote marks. This means that whatever scanner is used must treat the sequence

quote-mark bracketing-character quote-mark

as a special case, to be interpreted as meaning the bracketing character standing alone.

Convention 2. Provide a command which temporarily deprives a certain character of its status of being a control character, and possibly substitutes another character; the status quo to be restored by the execution of a countermanding command. For example, to deal with the string abc"def as a quoted parameter, let " be temporarily replaced in its control role by '. Now we can have the quoted parameter 'abc"def' without misinterpretation. Second example: Suppose that we wish the string abc,def to appear

in a list of parameters, where comma is the usual separator. We could, for the purposes of interpreting this list, declare blank to be the separator character. Then we could write

F (ADF abc,def ghi)

and have abc,def properly recognized as the second parameter.

Both of the conventions mentioned here require explicit provisions in the language processor to handle them.

There is another problem, closely related to the control character interpretation problem, called the "matching brackets problem," which is discussed in the next section. Solutions to the matching brackets problem are automatically solutions to certain aspects of the control character interpretation problem.

The matching brackets problem. The matching brackets problem is concerned with control characters which are normally used only in matching pairs: these are quotes, round brackets, (parentheses), square brackets, and curly brackets. The problem is concerned with how to decide which brackets form matching pairs. If there is a convention for denoting an individual unmatched bracket, and if all other brackets are present in properly nested form, then there is no problem in finding which brackets form pairs. However, if there is an error, in that a member of a pair is erroneously missing, or through some transposition the brackets are not in properly nested form, then the pairing cannot in general be discovered by inspection alone. To provide for discovering and dealing with such errors, some facility is needed for uniquely discovering which brackets form matched pairs. There is a convention which can be used and which will serve except perhaps in rare cases. This convention is one that can be brought into play at the user's option but it need not be "built in" to a language and processor system. This convention is simply a rule for forming an unlimited number of different bracketing expressions. The rule is this: let a left-bracket expression be any string (1) beginning and ending with a blank, (2) not containing a blank, and (3) containing one left parenthesis. A right-bracket expression will be a similar expression except it contains one right parenthesis. Examples of such bracket expressions are:

abc( )abc

a(bc abc)

It helps readability if the expressions have the parenthesis at the inner ends, but this is not a necessary restriction. It is easily seen that we have an unlimited stock of convenient brackets

```
1(      )1
2(      )2
793(    )793
```

A complicated function call written using this matched bracket convention might look like this:

```
func 1( A,tw 2( f          3( j,m )3  ")",r )2 , )1
```

If an error occurred, which resulted in the bracket)2 being lost, a scan of the set of parameters of funct would show immediately that an error existed and the procedure call mechanism would not be prevented from continuing its analysis of the expression in which this function call to funct was embedded.

What is a formal parameter? It is appropriate for us to examine carefully what a formal parameter is, and how it behaves. It is commonly agreed that a formal parameter is a placeholder (in a prototype); something is to be substituted for this placeholder in a copy of the prototype. "Prototype" in this context of course means "procedure body", and "substituted ...in a copy of the prototype" means that a call of a procedure has this effect, whether or not it is actually carried out that way.

When we consider further the actual nature of the placeholder we find that "placeholder" can be interpreted in a variety of ways:

1. As an individual parameter in a structured string: that is, as a given parameter in a transformation call. Example: Consider a program statement in our standard prefix form: T(X,Y,Z) Any of the names T,X,Y, and Z can be regarded as formal parameters, to be replaced under some set of rules specified elsewhere.
2. As an individual parameter appearing possibly more than once in a partially-ordered set of structured strings (in this context, a procedure body). We can forget for a moment the complication of allowing partially-ordered sets of strings; the procedure bodies that we normally think of are simply linear strings. This role of parameter is the one that applies to conventional procedure calls in ALGOL, FORTRAN, PL/I and the like.
3. As one or more appearances of a substring in an unstructured string. This viewpoint allows a more general application

of the formal parameter concept. In fact, being able to substitute one substring for another is the most general kind of string substitution that we can envision.

4. As an identifier associated with one or more components of a data structure, which structure may later be interpreted as a procedure.

A placeholder of the third type mentioned above is fundamentally different from types 1 and 2. Type 3 obviously requires a scan of the procedure definition at call time (that is, at the time a procedure is invoked). Notice that type 4 could be used to avoid the need for this scan, by making it possible to "name" the insertion points, which would perhaps be initially represented by null substrings.

While the discussion of formal parameters has here centered about procedure definitions, it should be remembered that the concept of formal parameter applies equally well to data structures, for in fact procedure definitions are simply a specific form of data structure.

Formal and actual parameters. There is no real need to distinguish between formal and actual parameters as far as how identifiers for them are constructed. The feature which distinguishes a formal parameter from an actual one is the fact that the formal name is systematically replaced at some point by an actual name. This substitution must be made with due regard for the presence of local data elements ("local variables"), otherwise the prototype may be rendered unusable for a succeeding substitution (because, as the logicians would say, of the confusion between bound and free variables). Consider the prototype procedure:

```
proc P1(a,b);
local x;
x<----a;
a<----b;
b<----x;
end P1;
```

In the above, x is bound (is a "dummy variable", and it could have any arbitrary name) while a and b are "free". If we give the call

```
CALL P1(x,y);
```

then a copy of procedure P1 is made and in this copy x replaces a and y replaces b, yielding:

```
proc P1(x,y);
local x;
x<----x;
x<----y;
y<----x;
end P1;
```

which is not what is wanted, and which will not yield the desired result.

If the prototype itself is to be manipulated by the program, then calls of the procedure should make substitutions in a copy of the prototype, leaving the prototype intact. Otherwise names may be arbitrarily changed by the substitution process, thus invalidating later references to the prototype for purposes of manipulation.

Several situations can be described which exemplify the difficulties of making a distinction between formal and actual parameters:

1. A procedure P1 operates on a procedure P2, patching sections of P2 together to make a new procedure P3. A call is then made to P3.
2. A procedure P4 when first called tailors itself to be more efficient in response to the class of calls exemplified by the current call. After tailoring itself, it closes off the tailoring process from further use, and then proceeds to do its regular work.

The common characteristic of these problems is the dual nature of some of the parameters, parameters which at one time are considered to be actual and at another time are treated as formal.

Reinfelds (in his paper "The Call-By-Symbol Concept: A symmetrization of the Scope of Variables in Actual Parameter Expressions of Subroutine Calls", submitted for the 1968 FJCC) advances the interesting concept that in a procedure call one may wish to express formal parameters. One can in this way achieve the effect of a call by name but at the samt time prevent side effects. In order to indicate that a formal parameter name is being used in the procedure call, some special indicator must be used. Reinfelds advocates that the declaration symbol x appear in the parameter list indicating that "x" in the parameter list is a formal parameter of the procedure being called. My approach of making no distinction between actual and formal parameter identifiers, however, renders Reinfelds' concept of no use in this context.

Parameter-passing mechanisms. The task to be considered is the means of "passing" to a procedure parameters given in a call to that procedure.

We need to be able to access each actual parameter, knowing which formal parameter we want the correspondent of. We should be able to ask for an actual parameter either by its position in a call, or by its keyword parameter name if it has one. To do this we must be able to reference the "current call of the procedure P". It is not sufficient simply to say "the current call" since at the time of referencing other calls and other procedures may have intervened. In the case of recursive calls one must be able to access the call"n" levels back, where n will frequently be "1".

I had originally thought of setting up an explicit correspondence table between formal parameters and actual parameters. This table would have to be filled in either automatically by the call mechanism or by statements explicitly given in the body of the procedure. A problem arose with trying to define the structure of this table, for some formal parameters may have the role of identifiers but other parameters may simply be uninterpreted substrings.

Upon further consideration, it didn't seem that explicit construction of this table was necessary. Each entry in the table would in general be referenced only once, as each actual parameter was picked up to be treated. It seems simpler to have a primitive which accesses an actual parameter corresponding to a given formal parameter. This will be discussed in more detail below.

There must be a way of matching up actual parameters with formal parameters. There are actually two standard ways of accomplishing this:

Here are some illustrations of simple procedure calls:

1. Call with "positional parameters", that is, the correspondence between the actual and formal parameters is known because the actual parameters occur in pre-specified positions (in the parameter list). An example of a call having positional parameters is:

   REMAINDER(A,B)

   In the definition of REMAINDER there occurs implicitly or explicitly the information that the first actual parameter corresponds to the first formal parameter, and the second actual

parameter corresponds to the second formal parameter. The formal parameters may have explicit identifiers. They might be identified as x and y respectively. Then the parameter matching process would say that x corresponds to A and y corresponds to B. However, they need not have explicit identifiers. Assuming that we have a way of referencing the first actual parameter, in a way that we have not yet discussed, the definition of REMAINDER might use the expressions "actual parameter #1" and "actual parameter #2" in the statements to be executed.

2. Call with "keyword parameters", that is, the correspondence between the actual and formal parameters is specified by explicitly giving pairs, each of which consists of a formal parameter followed by its corresponding actual parameter. An example of a call having keyword parameters is:

DISP (p2=M,p4=S,p1=KT)

Notice that in such a call an actual parameter need not be given to correspond to each possible formal parameter, and the actual parameters which are given do not need to be given in a specified sequence.

An additional problem of matching formal and actual parameters arises, however, where a transformation is defined for a set of parameters of the same type but the number of the parameters in the set is not fixed. An example might be SUM ( ) which could easily be defined to compute the proper result for an indefinite number of arguments. What we have here then is a function defined on a set of arbitrary size, where the parameter to be passed is not an identifier of the set but rather an enumeration of its members. For functions which are defined in algorithmic form, it is of course possible to write the procedure so that it examines the parameter list and computes the function properly for the number of arguments exhibited as actual parameters.

If the transformation procedure knows whether to expect a list of parameters, or only the identifier of such a list, explicitly following the transformation identifier, then there is no ingenuity required to pick up the individual parameters properly. If we wish to be able to give either, that is, interchangeably, the explicit list or the identifier of such a list, then the transformation procedure must be able to examine the parameter actually delivered and decide how to handle it.

There is a variation on the format of a procedure call: namely, instead of explicitly listing the relevant parameters, giving a reference to a parameter list. This list may, in turn, be either a single list of positional parameters, to be matched 1 for 1 with the formal parameters, or it may be a list of keyword-parameter pairs, matching actual and formal parameters. Example: let f(a,b) represent a function f with formal parameters a and b. Let z be a list of two actual parameters x and y. The function call might then be written f(z), where the function definition will recognize that the type of z is list, and that when a parameter of type list is presented, it must be decomposed into its constituents and these constituents used as the parameters.

There is a question as to whether this should be caller's option or procedure definer's option. If it is the caller's option, some special symbol is needed to indicate the use of this option, and some extra mechanism in the call interpreter to test for this option. If it is the procedure definer's option, there is no need to provide any special mechanism; the procedure writer is responsible for testing the type of the parameter and taking appropriate action.

Parameter interpretation. There are several fundamentally different ways to interpret a parameter which appears in a procedure call. The first issue is whether or not the parameter is to be interpreted at all. We must decide between the following two cases:

1. The parameter is a string which is to be substituted without interpretation at this time. This is often called "call by name". A more suggestive term is Strachey's term "call by substitution", and it is the term I will use henceforth.
2. The parameter is a reference to a data element. This is called "call by reference".

The processor must be told whether a given parameter is to be interpreted or not. In principle this decision could be made on the basis of the form of the parameter, but this would mean a lack of flexibility, since in the call by substitution we want to be able to substitute an arbitrary string. I conclude therefore that the indication of whether a given parameter is called by substitution or called by reference must be explicitly given in the called procedure.

A parameter to be interpreted is a reference to a data element. The types of such references are two:

1. A string to be interpreted as naming a data element. While it may be a complex name, we can assume that the interpretation of this name is a standard process. I call this string a "data reference".
2. A string with or without an explicit set of parameters in parameter brackets, to be interpreted as a function call. I call this a "command reference".

The intellectual distinction I have made between data reference and command reference is really only a matter of emphasis, for in the case of a data reference we are in fact invoking a function. This function is the standard naming function. It is, for example, the function which, given an expression such as PR[3], accesses an element of the array "PR". We will see later that the naming function logically must be called implicitly rather than explicitly.

Notice that a consequence of my philosophy is that there is no concept corresponding to the ALGOL call-by-value. Call by value yields a theoretically unnamed, unstored value, and my approach says that there can be no such entity, that every value-string used as a parameter is stored and has at least one identifier.

The problem at hand is how to distinguish a data reference from a command reference. We must do it either by inspecting the form of the parameter or by referring to information external to the parameter itself.

This need to distinguish between a reference which accesses a data element and a reference which invokes a function is seen in the use of two sets of brackets in ALGOL.

A[N] means access the Nth element of the array A. Note that this includes a label array (a switch).

A(N) means invoke the function A with the parameter N.

The shape of the brackets indicates which kind of access. Where there is no parameter list in brackets, however, the two kinds of reference cannot be distinguished.

A fundamental problem of ambiguity crops up here, in the case of a parameterless function call. The call appears as a string. The problem is that we cannot tell by inspection whether it is to be referenced (given

a "data interpretation") or to be invoked (given a "command interpretation"). Some languages avoid this ambiguity by requiring the parameter brackets to be present, thus distinguishing the function call, and always giving the function call a command interpretation. In such languages, of course, the function can never be interpreted as data, which is an undesirable restriction.

Here is an illustration of a parameterless function call. Assume that T1 and T2 are the identifiers of two parameterless function procedures, each procedure yielding an integer result.

SUM(CALL(T1),CALL(T2)) computes the sum of the results of executing T1 and T2.

SUM(T1,T2) is undefined because SUM is not defined for the data type of the procedures T1 and T2 (distinct from the data type of their results, which is integer).

CONCAT(T1,T2) is defined; the result of the string obtained by concatenating the string named T1 with the string named T2.

Most programming languages resolve this ambiguity by establishing some arbitrary and restrictive rules. For example: a designator of a parameterless function procedure is indistinguishable from an identifier of a data element. Grenoble ALGOL requires such a function procedure designator to be enclosed in parentheses in order to force its execution. FORTRAN avoids the problem by requiring function names always to be followed by a parameter list, even if it is a "dummy" list. PL/I determines if the parameter is the name of a function, and, if so, executes it. None of these languages, and in fact none of the languages with associated compilers, permit a function definition to be referenced as data, that is, as a string.

A second example is where a parameter consists of an identifier followed by a list of parameters enclosed in parentheses. This could be interpreted either as a function call, in which case a command interpretation is probably wanted, or as a compound identifier such as an array element name, in which case a data interpretation is probably wanted. ALGOL avoids the problem of distinguishing these two uses by requiring that in the second case, special subscript brackets (square brackets) be used. FORTRAN and PL/I determine whether the parameter names a function or a compound data element and give the parameter a command interpretation or data interpretation accordingly. As before, these interpretation rules are

fixed and therefore do not permit executing a function procedure stored in an array of procedures, nor of treating a procedure as a data element.

It is often convenient to write a literal as a data reference. That is, instead of writing a data reference we may wish to write instead an actual value-string. If we wish this feature, then the processor must be arranged to recognize when we have done so, create a storage cell for the value-string, and create an identifier for it. The value of the parameter is the identifier thus created. A frequently-used convention is to enclose the literal in quotation marks, and I adopt that convention.

To summarize, we can have the following forms of reference expressions:

Form	Meaning	Interpretation
A	identifier of data element	data
A	identifier of parameterless procedure	command or data
A(P,Q)	function procedure designation with parameters	command or data*
A(P,Q)	identifier of component data element within a composite, or a set of arguments of a function represented as a function table	data
"A"	the value-string "A"	data

* We might choose not to define what this expression means.

Note that in general we want to be able to give any one of the five forms of references. We would not, in general, wish to resolve ambiguities by giving the type of interpretation (data vs. command) in the parameter treatment specification, because we want the flexibility of using these types of parameters interchangeably. Hence the distinction between command references and data references must not be made on the basis of parameter inspection alone.

One solution to the problem would be to label each parameter as to whether it was a command reference or a data reference, such as CALL(Y) or DATA(X), respectively. This is somewhat clumsy. What is worse, it is technically inconsistent, as will become apparent in the later discussion of the fact that the data referencing function cannot be an explicit one. The remaining question is how to simplify this clumsy convention of CALL(Y) and DATA (X). We could, for example, dispense with DATA ( ) and

use simply X for a data reference. Only if X had the form CALL (Y) would we have it receive a command interpretation instead.

So, for example, to compute the value of the formula (which we assume appeared as an actual parameter)

A + random integer

where the random number is obtained by a parameterless function procedure, we would write

CALL(PLUS(A,CALL(RANDOM));

If we allow ourselves the simplification of omitting parentheses surrounding a one-member parameter list, or at least after the expression CALL, we could write this more readibly as

CALL PLUS(A,CALL RANDOM);

Some users may regard as inconvenient the inclusion of the word CALL in each parameter whose execution is invoked. Such users may choose instead to specify in the definition of a transformation a fixed interpretation of the parameter: he can specify either "access" or "execute". The price to be paid for this convenience is the lack of flexibility in being able to have the mode of treatment deducible from the form of the actual parameter.

As a notational convenience we could easily define the CALL statement to accommodate multiple parameters; that is, for example,

CALL A,B,C;

could be used to mean call A then call B then call C. Each call could have a parameter list as well; thus we might write

CALL A(X,Y),F(Z);

I have suggested here one reasonable set of conventions for distinguishing command interpretations from data interpretations. It should be possible for the user to change readily whatever conventions might be established initially for a given processor.

Not all the possible complications have been resolved. For instance, it could happen that both a parameter list and a subscript list occur in the same parameter expression. For example:

T(A) (B,C)

might mean "select the function with index A in an array T of function definitions and use the function with the parameter list (B,C)". If this construction is to be allowed, then we need to have a way of defining the

convent' n to be used for interpreting it. Further we must keep in mind that we might want a data interpretation expressed simply

T(A)

in order to access the function definition as data. Hence whatever conventions are established should allow for the proper interpretation of the latter expression.

The data referencing function—a necessarily implicit function. Notice that obtaining the identifier of an element whose data reference is given is itself the performance of a function. It is, and necessarily must be, a function which is not expressed explicitly but is rather implied. Suppose a function of one argument:

f(a)

If we wanted to show the data referencing function i explicitly, we might write

f(i(a))

But in doing so we have not been consistent, in that the argument of the function i is the "bare" data reference a. We cannot overcome this no matter how many times we replace the inner a by the more explicit i(a). We are forced to realize that an innermost parameter must be interpreted as a data reference expression, of which we are usually to take the identifier. If the data reference expression is a single identifier, then its interpretation is simply that identifier.

This leads us to a related issue, that of indirect naming.

Indirect addressing or naming. It may on occasion be useful to refer to a data element by an "indirect name", that is, by giving an identifier of an identifier of a data element. This feature is often called "indirect addressing" when speaking of an address of an address of a machine word. Explicit indication of indirect naming in SNOBOL is given by the prefixing to a name the symbol $. This symbol can be taken to be the function "evaluate" in the conventional sense. Example: let "1" be the contents of a simple data element identified by the string A. Let "A" be the contents of a simple data element identified by the string B. If we write as a parameter

F(A)

this will be interpreted as

F("1")

that is, the function F applied to the value "1". If we write as a parameter

F($(B)) or F(eval(B))

this will also be interpreted as

F("1")

In this latter case, the parameter is effectively being evaluated twice in succession. The first evaluation is called for by virtue of the function F calling for the normal evaluation of its parameter. An evaluation of the result of the first evaluation is called for by the "evaluation function" designated by either one of the devices exhibited above. SNOBOL uses the symbol $ as the name of this function, so that

$A1

in SNOBOL means the data element whose identifier is the value of another data element whose identifier in turn is A1. Such expressions can be nested to any practical depth to achieve multiple-level indirect naming, for example

$($A1)) $($($A1)))

Parameter types. The matter of concern is the "types" of parameters associated with transformations——both input parameters and an output parameter or parameters. "Type" requires some definition. Where a parameter stands for a simple data element, the parameter type is the name of the domain of that element. Where a parameter stands for a composite data element, the concept of parameter type embraces structure class names (such as "list", or "company") and perhaps also the domain names of its components. Thus, a parameter type specification of a composite element might be "list of integers", or "set of 4 x 4 arrays of integers", or "set of arrays".

Every transformation is necessarily defined only for some specific choice of parameter types. That is, if a parameter in a call is not of a type specifically provided for by the transformation definition, execution of the transformation will "fail". What "failure" means is discussed elsewhere under "Invalid transformations". To illustrate these concepts, consider that a function "sum" could be defined for a variety of parameter types: possibly any mixture of values from the domains of integer, real, and fraction. It might be defined also for strings: in this latter case it would probably mean "concatenation". But the function

would probably not be defined for a mixture of strings and integers. To repeat, a given transformation is necessarily defined for one or more fixed combinations of parameter types. If the transformation receives as input any other combination of parameter types, it will not give a correct result. In such a case the transformation must not give a result which could be misleadingly interpreted as correct; that is, it must explicitly signal when such a failure occurs. Whether a transformation is to avoid the computation of invalid results ("garbage") or not is up to the person who writes the transformation definition.

It may happen that a parameter is not from a domain (or of a type) specifically anticipated by the writer of a procedure. It may also happen that the given parameter value has an equivalent in a domain which is acceptable. Example: Let there be a procedure which computes area, given dimensions in feet. What should happen if a call to this procedure has parameters given in inches? It would be desirable to have a convenient mechanism whereby the writer of the procedure could ask to have, in such a case, a search made automatically to find the corresponding values in feet and use these converted values to proceed with the computation. Note that invoking such a mechanism should be at the writer's option; it would be unnecessarily clumsy to always invoke such a search automatically.

The writer of a transformation definition (procedure) needs program statements (calls to transformations, probably primitive ones) which enable him to determine the types of his input parameters, to check the types to verify if they are acceptable, possibly to convert the values to equivalents in an acceptable domain, and to signal whether the transformation has been successful (or, if not, what the nature of the failure was).

Substitution of parameters. Having picked up an actual parameter and treated it, we must next specify how and where the result is to be substituted. It may be substituted for each appearance of a given character (or substring) in the procedure body, which will require a full scan of the procedure body. Or it may be substituted as the "value" of a named part of the structure which constitutes the procedure body. The choice of these two substitution methods must be specified in the procedure body, or in the call mechanism (that is, in the interpreter).

Invalid transformations. As mentioned earlier, it may happen that

part way through the execution of a transformation a parameter may be found to be of the wrong type, or the argument specification may be in some other way not satisfied. Even if the arguments (parameters) are proper, it might happen that the transformation cannot be correctly completed. There arises the question of how to know the precise effect of the transformation attempted under such circumstances. Since side effects can occur, the effect of an invalid transformation must be known to the user in order for him to be able to recover.

There are two extreme choices for the effect of an invalid transformation: (1) no action, and (2) all possible actions up to the point of invalidity. The no-action case might be preferable; in a sense it is a cleaner alternative. In some cases, however, it is extremely difficult to achieve because it can mean having to remember an arbitrary amount of processing in order to put things back the way they were before the transformation began. The all-possible-legal-actions case seems wasteful, in that some unnecessary actions may be done, but it is much simpler to explain, comprehend, and remember.

Another choice is to let the user in general define how he wants invalid transformations handled by the way he defines a given transformation. He can, for example, check to see that all arguments are legal before transformational changes are executed.

There is a need for some standard error signal mechanism. Earlier I discussed the use of a result value-string of "UNDEFINED" as a way to indicate failure of a function. We need a similar convenient convention to indicate the failure of a transformation which is not a function. Some of the possibilities for such a convention are:

1. A success-failure flag to be set by each transformation. If there is only one such flag, however, confusion will arise in the circumstance of parallel execution.
2. A success-failure first-in-first-out queue, on which is entered a transformation name and success-failure flag (and perhaps other identifying information such as an identifier of a statement invoking the transformation). This queue would be a standard data element accessible by conventional referencing techniques. An alternative would be to have this queue maintained in the processor in an undefined way, and to have it be able to be referenced only through some primitive function specifically designed

for that purpose.

3. To pass success and failure return point identifiers ("labels") as parameters to each transformation.

Procedure call mechanism. We have discussed at length the actions and requirements of procedure calls. Now we must be more specific as to how procedure calls are accomplished. Assume that control has been transferred to (a copy of) a desired procedure. The actions that can occur which are related to the calling process, as distinguished from the "work" the procedure is called upon to do, are:

1. Locate the call.
2. Select an actual parameter from the call.
3. Treat this parameter according to some prespecified method.
4. Optionally, check the type of the parameter to varify that it is acceptable to this procedure.
5. Optionally, if the type of the parameter is not acceptable, seek an equivalent in a specified domain.
6. Substitute the result of the parameter treatment into the procedure body, according to some prespecified substitution rule.
7. Repeat steps 2 through 6 for all parameters required.
8. Upon exit from the procedure, set an indicator to "success" or "failure".

We now proceed to discuss these actions in more detail.

The need for locating a specific call seems at first unnecessary. Why should it not always be the most recent call executed? The answer is that the parameters need not be picked up, processed, and substituted immediately upon entry to a procedure; in general we want to be able to process them anytime during the execution of the body of a procedure, and perhaps in some cases certain parameters will not be needed and therefore need not be processed at all. So we may find that in procedure A we execute first a call to procedure B, and then wish to process parameters of the call to procedure A. Another possibility arises in a recursive procedure, where at some point we may wish to process a parameter in the current call, and at another point we may wish to process a parameter in a previous call.

Having located the desired call, we next need to be able to select from it a desired parameter. It may be selected on the basis of its

position in the list of actual parameters, if it is a list of positional parameters. Or it may be selected on the basis of its name or keyword accompanying it in the list of actual parameters. (Keyword parameters can be considered a special case of a more general concept: that of picking parameters on the basis of their partial contents. That is, we could select a parameter they begins KEY= , or one that ends in S, or one that contains at least two asterisks, etc.) Naturally it can happen that the sought-for parameter is absent, in which case provision must be made for indicating this fact to the parameter-pickup mechanism (a function).

Next the parameter is interpreted. We have discussed earlier the standard interpretation methods. For flexibility, however, the user need not be confined to these standard methods. He should be able to devise his own.

A check of the "type" of a parameter means to inspect the parameter and determine if it has the characteristics assumed by the procedure body. If the parameter is expected to be a simple data element, then the check would probably be to verify that the parameter value was from the proper domain. In case a parameter is expected to be a composite element, a variety of checks are possible: to check that the composite element has the desired structure, to check also that the values of simple components are from specified domains, or to check that the element is of a given class (has a given model name).

Suppose that a parameter has been selected and that the type check reveals that the value is not from an acceptable domain. It would on occasion be desirable to call for a search for an equivalent value in a specified domain. For example, if a parameter delivered a value in feet for a procedure designed to expect inches, and if suitable equivalence tables or algorithms existed, a search could be expected to yield the equivalent value in inches. It is not essential that such a search be built into a system; the essential point is that the creation of such a search routine and its invocation as part of the procedure call mechanism should be possible.

Assuming that a parameter has been processed and found valid, the next step is substitution of the treated form of the parameter into the procedure body, replacing a formal "parameter". This is a non-trivial matter, however, in view of the earlier discussion about the alternative

interpretations of formal parameters. The substitution operator must specify the nature of the substitution to be performed, according to the assumed characteristics of the formal parameters. The substitution may be for a given parameter in a procedure body, or for a given substring, for a given named component of a composite data element, or for some other programmer-specified part of a procedure body.

There should exist some simple way to indicate the successful completion of a transformation. In the case of a function, which returns a value, the value "null" can often be taken as an indication of failure, although not always. In the case of a transformation which does not return a single value, however, we need some way to be able to determine whether the transformation procedure completed its assigned task successfully. I am not suggesting that this flag be set automatically, but rather that primitives exist for setting and testing such flags, and that the setting and testing be done at the option of the user.

It should be obvious that whatever transformations are needed for implementing this procedure call mechanism must be primitive. If they were not, then these elements of the procedure call mechanism would themselves invoke the procedure call mechanism, and a non-terminating recursion would occur.

Remark on multiple entry points. Procedures can have multiple entry points (though personally I think it is poor programming practice). Since we treat declarations as executable commands there is an important consequence related to multiple entry points: that is, that the thread of control from each entry point must "pass through" the "declarations" that are going to be needed in the body of the procedure. This can be done fairly simply by packaging the set of declaration statements as a subprocedure and calling it immediately after entry at any entry point.

CONTROL SEQUENCING

This section is concerned with how the processor is told the sequence in which transformations are to be executed.

The "thread of control" concept. Nearly all sequence controls in current use are based on a "thread of control" concept, by which is meant that the majority of transformation calls have a successor defined by convention. The advantage of this technique is that control sequencing can usually be expressed by physical arrangement of transformation calls

rather than by the successor of each transformation call having to be explicitly given. We therefore provide for this thread of control concept. The user is not required to use it for user-defined transformations, however; he may instead define all his transformations with a parameter which explicitly specifies the successor.

Some elaboration is in order concerning the physical arrangement of transformation calls. Each transformation call is a data element interpretable as a transformation name plus a set of actual parameters. The standard form of a transformation call is a simple data element whose contents is a value-string. An alternate form which may be useful is a composite, the first component of which is the transformation identifier and the remaining components of which are the actual parameters. The successor of any given transformation is explicitly tied to that call by a successor relationship.

A thread of control may split, or "fork", into two or more threads, which requires the creation by the processor of parallel asynchronous paths of control. Several such threads of control can merge, or "join", into one; control must reach the merge point from all merging threads before control proceeds forward from that point. Every thread of control eventually terminates, either in a join, or at some transformation which stops further processing of that thread.

Since the successor relationship as used here is not a transitive one, loops are not prohibited. Hence in the most general case, the paths of control may form a network, which I will henceforth refer to as a"control network", or a "network of control".

The sequence control mechanism. Our concept of the sequence control mechanism is that of a processor, defined as a primitive, which receives its commands via primitive transformations. It provides for parallel asynchronous threads of control as well as for the initiation of execution sequences whenever specified conditions become true.

Part of the sequence control mechanism is a set of control lists, which are"ordinary" data elements with known identifiers. These data elements are created as needed, one for each thread of control, and destroyed when no longer needed. Each such data element is a composite; it is in effect a pushdown list which stores a list of identifiers of return locations. Since these data elements are ordinary ones, the user has access to them and can inspect or modify them as he wishes. In order

for the user to know the identifiers of these control lists, they must be identified by some standard naming scheme. In order for the user to know the identifier of a control list of a particular thread of control, a primitive transformation THREAD-IDENT executed in that thread of control will yield the identifier of the associated control list.

Arbitrary complex sequencing. Successor relationships, iteration controls, and branch instructions will provide for the majority of sequencing needs. When more complex sequence controls are needed, however, they can be written as procedures. The general form of such a procedure is a loop. In the first part of the loop is determined the identifier of the transformation to be executed next. In the second part of this loop, this identifier is substituted in a call expression and the specified transformation is thus invoked. The thread of control usually returns to the beginning of this control procedure, though on occasion it may terminate. This technique of programming sequence control rules may find applicability in applying various priority control algorithms [See Gorn, 1959]. It can also be used in controlling iterations, and in following explicit "chains" of identifiers, where the successor relationships have been expressed between elements which contain identifiers of transformation calls rather than the calls themselves. It is interesting to note, and it may be of considerable usefulness, that a user program can have all the transformation calls which "do useful work" written as individual and independent data elements without successor relationships between them, while all the sequencing of execution can be controlled by a separate procedure.

Transformations of sequence control. Transformations which are concerned with control sequencing are presented in the next chapter along with the other types of transformations.

CHAPTER 6. BASIC TRANSFORMATIONS

What follows is a list of manipulations and functions which illustrates the nature and variety of transformations which are needed in the system being developed here. This list may not be complete, as it has simply been 'thought up' and has not been tried out in real-life situations.

BASIC TRANSFORMATIONS AS DATA

Having developed a theory of data elements and values, and having discussed the framework of transformations, we are now in a position to define a set of basic transformations of data. To the extent that domains are defined by enumeration, transformations of domain will also turn out to be transformations of data. As we have just seen in Chapter 5, transformations of sequence control are largely transformations of data.

Transformations of data can be divided into two broad classes which may overlap slightly:

1. Transformations which perform creation, destroying, selection, and testing of identifiers, cells, and composites.
2. Functions which take value-strings as arguments and have a value-string as result. A special sub-class of such functions consists of those which operate on the strings which are transformation calls.

The description of a transformation will follow the general format.

1. Name of the transformation or function, including parameter list.
2. fect of the transformation.
3. Exit-value resulting when the transformation appears as a parameter of another transformation. The exit-value is always an identifier. it is either an identifier of a valid result (which is a data element), or an identifier of a cell containing the value-string UNDEFINED or NULL.
4. Illustration, if useful.
5. Comment, if applicable, including a mention of when a transformation is readily definable in terms of other transformations.

Create a cell.

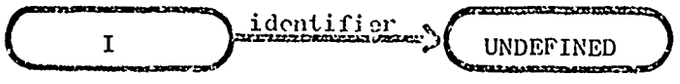
Name: CREATE-CELL, or CREATE-CELL(I)

Effect: To generate a new cell and associate with it an identifier. If I specifies or references an identifier, then that identifier will

be associated with the new cell. If no such identifier is given, then the processor will generate one.

Exit-value: The generated or specified identifier of the new cell.

Illustration: The result can be diagrammed:



Comment: CREATE-CELL(I) can be defined in terms of CREATE-CELL and ASSOCIATE-IDENTIFIER (to be defined later), thus:

CALL ASSOCIATE-IDENTIFIER(I,CALL CREATE-CELL);

Copy a cell.

This is a case of COPY ELEMENT, which see.

Assign contents of a cell.

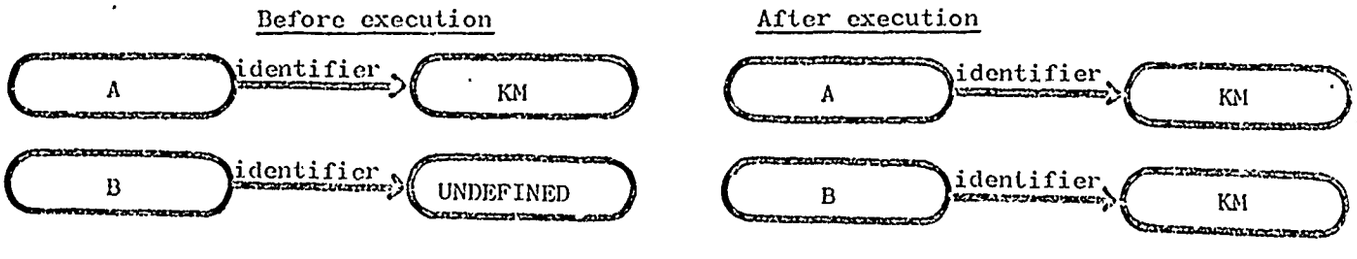
Name: ASSIGN-CONTENTS(I,E)

Effect: Replaces contents of cell or cells referenced by I by a copy of the contents of the cell referenced by E.

Exit-value: An identifier of the result cell. If I is an identifier, then the exit-value is precisely I.

Illustration: Consider the transformation call:

CALL ASSIGN-CONTENTS(B,A)



Comment: This transformation corresponds closely to the familiar "assignment statement" I = E or I ⟵ E.

Destroy a cell.

This is a case of DESTROY-ELEMENT, which see.

Associate an identifier.

Name: ASSOCIATE-IDENTIFIER(E,I), or ASSOCIATE-IDENTIFIER(E)

Effect: To associate the identifier referenced by I, with a data element referenced by E. If the second form is used, where the user does not specify the identifier, then the processor will generate one. If the identifier to be associated is not already in the identifier list, the processor will put it in the list.

Exit-value: The identifier associated with the referenced data element.

Illustration: To create a cell with an identifier of "X1":

CALL ASSOCIATE IDENTIFIER (CALL CREATE-CELL, "X1")

Comment: While a separate transformation could be defined to create an identifier there seems to be no need for it, since ASSOCIATE-IDENTIFIER creates an identifier at the same time a data element is created.

Dissociate identifier.

Name: DISSOCIATE-IDENTIFIER(I), or DISSOCIATE-IDENTIFIER(I,E)

Effect: To destroy the identifier relationship between the identifier referenced by I and the data element referenced by E. If no E is referenced, then all identifier relationships between the identifier (referenced by I) and all data elements are destroyed. If this transformation destroys all identifier relationships associated with the referenced identifier, then the identifier is automatically removed from the identifier list.

Exit-value: Null.

Destroy identifier.

A separate transformation to destroy an identifier is unnecessary in view of the actions performed by DISSOCIATE-IDENTIFIER.

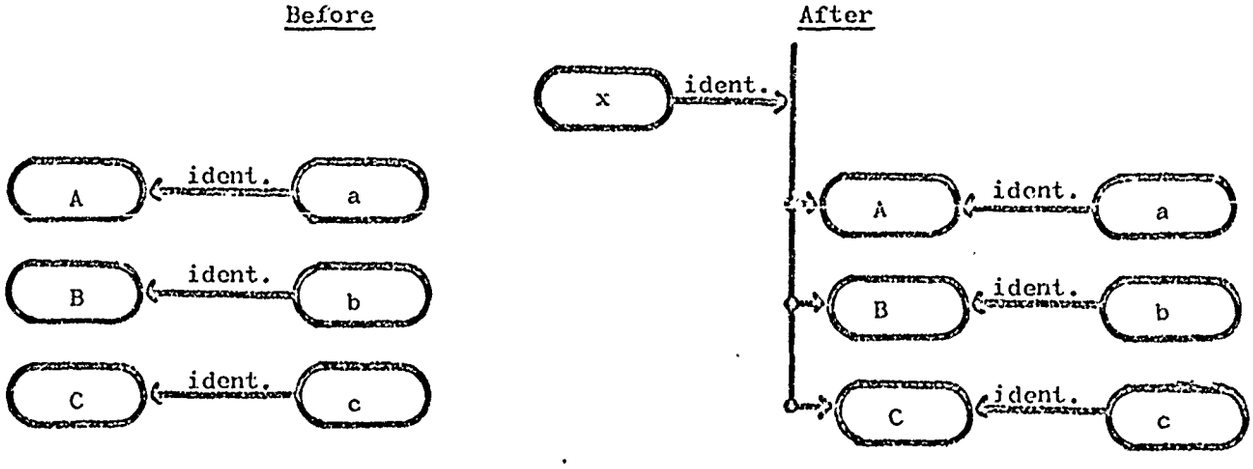
Associate components to create composite.

Name: CREATE-EXPLICIT-SET(E1,E2,E3,...)

Effect: To create a composite whose components are the referenced elements E1,E2, etc.

Exit-value: The generated identifier of the composite.

Illustration:



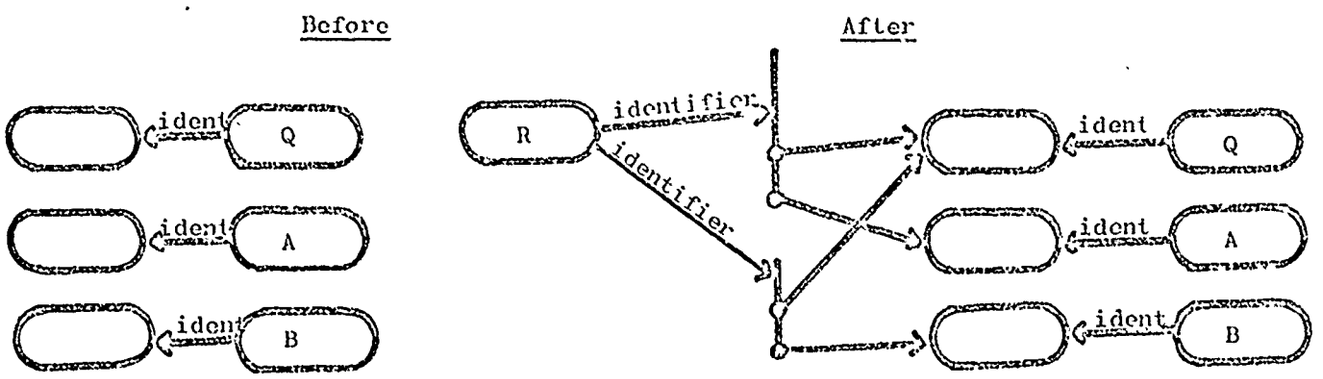
Associate components to create implicit set.

Name: CREATE-IMPLICIT-SET(Q,R,A,B,...)

Effect: To set up binary relationships QRA,QRD, etc.

Exit-value: Null.

Illustration:



Comment: This transformation is definable in terms of CREATE-EXPLICIT-SET, where each invocation creates one relationship(composite), thus:

CALL ASSOCIATE-IDENTIFIER("R",CALL CREATE-EXPLICIT-SET(Q,A))
CALL ASSOCIATE-IDENTIFIER("R",CALL CREATE-EXPLICIT-SET(Q,B))

Dissociate components of a composite.

To dissociate all components and destroy them, use DESTROY-ELEMENT, which see. To dissociate one or more individual components of a composite, use REMOVE-COMPONENT, which see. To dissociate all components of a composite without destroying them, use REMOVE-COMPONENT in an iteration over all components, thus:

CALL ITERATE-OVER-EXPLICIT-SET)I,J,CALL REMOVE-COMPONENT(I,J))

Destroy an element.

Name: DESTROY-ELEMENT(I)

Effect: Destroys the referenced cell or composite and all associations and relationships attached The components themselves are not destroyed. If there are identifiers uniquely associated with components of the element, these identifiers are destroyed also.

Exit-value: Identifier I of element destroyed.

Add a component.

Name: ADD-COMPONENT(I,C) or ADD-COMPONENT(I,C,N)

Effect: To include the element referenced by C as a new component of the composite referenced by I. If the parameter N is given, the new component is entered in the composite at ordinal position N.

Exit-value: Null.

Select a component by name.

Name: SELECT-COMPONENT-BY-PART-IDENTIFIER(I,P)

Effect: In composite referenced by I selects the component having the part-identifier referenced by P.

Exit-value: An identifier, generated if necessary, of the indicated component. If there is no such component, the result is undefined.

Select a component by position.

Name: SELECT-COMPONENT-BY-POSITION(I,N)

Effect: In composite referenced by I selects the component having ordinal position referenced by N.

Exit-value: An identifier generated if necessary, of the indicated component. If there is no such component the result is undefined.

Determine component position.

Name: COMPONENT-POSITION(I,P)

Effect: In composite referenced by I, searches for the component having the part-identifier referenced by P.

Exit-value: A generated identifier of N, the ordinal position of the specified component. If there is no such component, the result is undefined.

Remove component by name.

Name: DELETE-COMPONENT-BY-PART-IDENTIFIER(I,P)

Effect: In composite referenced by I, selects the component having the part-identifier referenced by P and deletes the component from membership in the composite. The component itself is not destroyed.

Exit-value: Null.

Remove component by position.

Name: DELETE-COMPONENT-BY-POSITION(I,N)

Effect: In composite referenced by I, selects the component having ordinal position referenced by N, and deletes this component from membership in the composite. The component itself is not destroyed.

Exit-value: Null.

Copy an element.

Name: COPY(E) or COPY(E,I)

Effect: Make a complete copy of the element referenced by E, except that a new main identifier referenced by I is assigned instead of copying the original identifier.

Exit-value: The identifier assigned to the copy.

Replace an element.

Name: REPLACE-ELEMENT(E,I)

Effect: First, destroy any data elements referenced by I. Second, make a complete copy of the element referenced by E and associate it with the identifier referenced by I.

Comment: Note the distinction between ASSOCIATE-IDENTIFIER, COPY, and REPLACE. ASSOCIATE-IDENTIFIER simply creates an identifier relationship between a specified identifier and a specified data element. COPY makes a copy of the specified data element and associates it with a specified identifier. REPLACE, which is analogous to ASSIGN-CONTENTS, is like COPY except that any data elements previously associated with the specified identifier are destroyed. It should therefore be obvious that REPLACE-ELEMENT is definable in terms of DESTROY-ELEMENT and COPY-ELEMENT, thus

CALL COPY-ELEMENT(E,CALL DESTROY-ELEMENT(I))

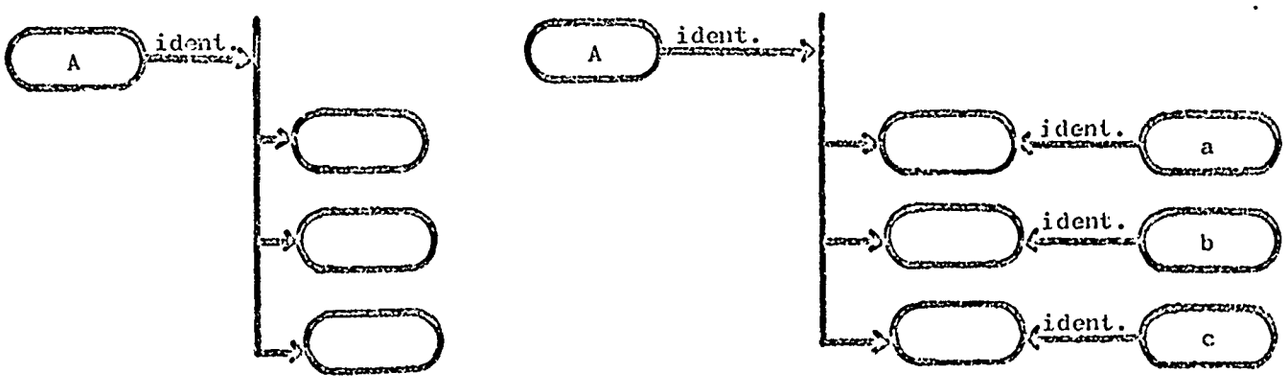
Cardinality of composite (of explicit set or sequence).

Name: CARDINALITY-OF-COMPOSITE(I)

Effect: To determine the number of components, N, in the composite referenced by I.

Exit-value: The number N. If there are zero components, the result is zero. If the composite does not exist, the result is UNDEFINED.

Illustration: Both of the following data elements have a cardinality of 3, because identifiers and metadata do not count as components:



Comment: If the composite is linear (that is, one-dimensional), then this function yields the length of the composite.

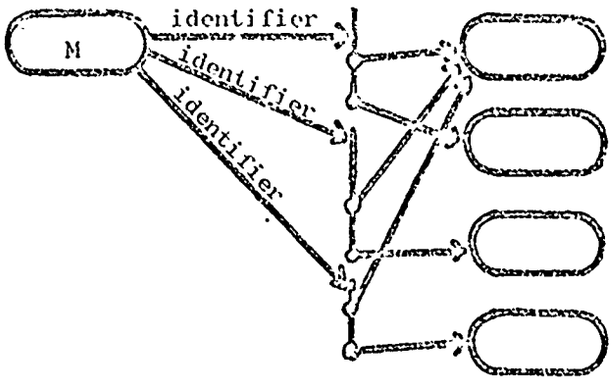
Cardinality of relationships (of implicit set).

Name: CARDINALITY-OF-RELATION(I,R)

Effect: To determine the number of elements, N, which stand in the relationship R to the element referenced by I.

Exit-value: The number of elements, N. If there are none, the result is zero.

Illustration: The cardinality of the implicit set M below is 3.



Comment: CARDINALITY-OF-RELATION could be defined in terms of cardinality of composite which results from applying the transformation CONSTRUCT-EXPLICIT-FROM-IMPLICIT and then taking CARDINALITY-OF-COMPOSITE.

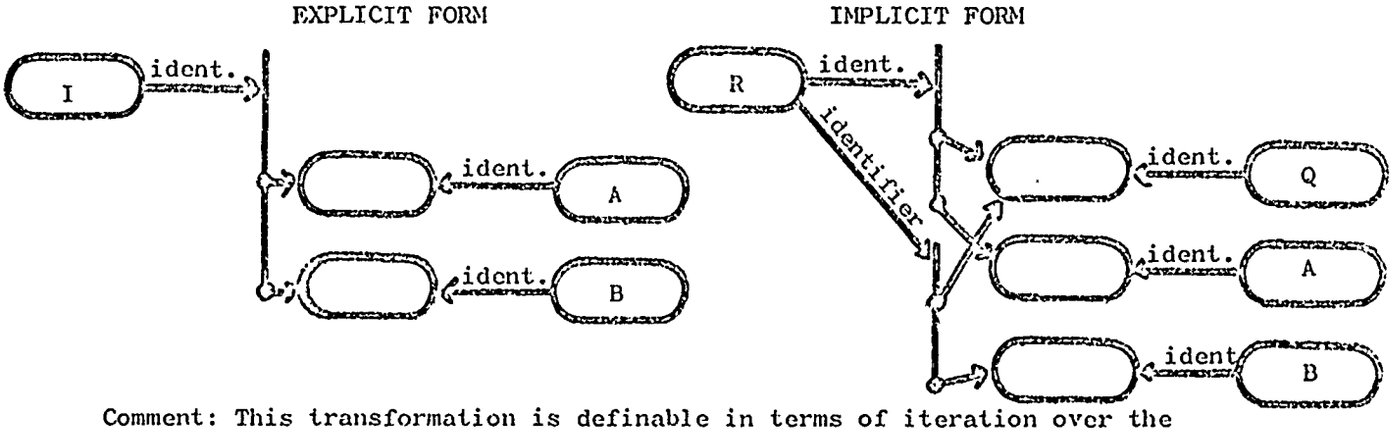
Construct implicit set from explicit set.

Name: CONSTRUCT-IMPLICIT-FROM-EXPLICIT(Q,R,I)

Effect: To set up binary relationships QRA, QRB, etc., where A,B, etc. are components of a composite referenced by I. Q references the identifier of the left member of the relationship. R references the identifier of the relationship itself.

Exit-value: Null.

Illustration:



Comment: This transformation is definable in terms of iteration over the components of the composite referenced by I, thus:

CALL ITERATION-OVER-EXPLICIT-SET(I,C,CALL CREATE-IMPLICIT-SET(Q,R,C))

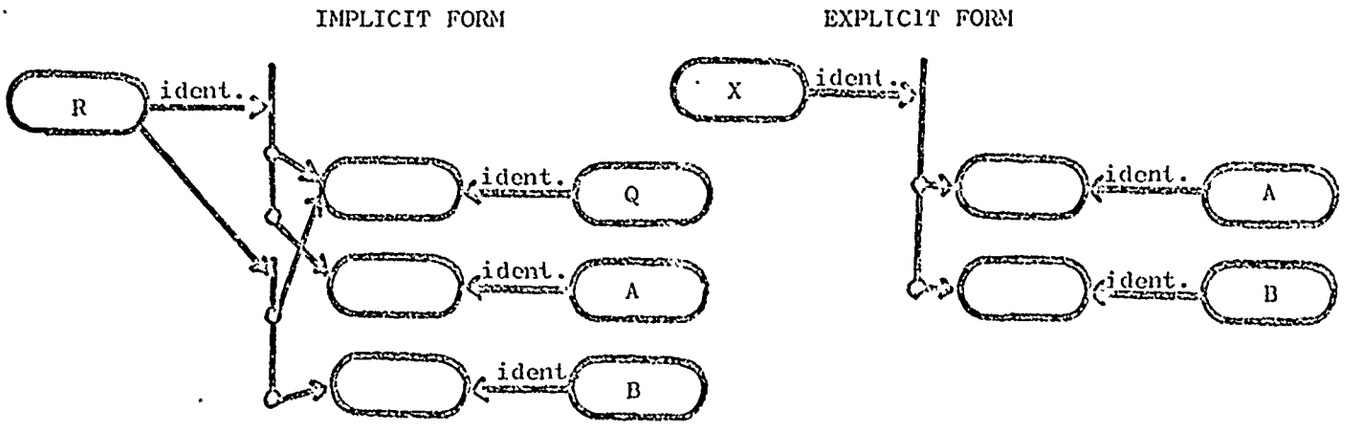
Construct explicit set from implicit set.

Name: CONSTRUCT-EXPLICIT-FROM-IMPLICIT(R)

Effect: To create an explicit set from the right-hand members of a relation R.

Exit-value: The generated identifier of the composite.

Illustration:



Convert value-string to composite.

Name: CONVERT-VALUE-STRING-TO-COMPOSITE(V)

Effect: Creates a composite in which the components are individual cells each containing a character of the value-string referenced by V.

Exit-value: An identifier of the created composite.

Convert composite to value-string.

Name: CONVERT-COMPOSITE-TO-VALUE-STRING(C)

Effect: Creates a value-string corresponding to a composite, referenced by C, which has a linear sequence of components, each component of which is a cell containing a single character.

Exit-value: An identifier of a cell containing the created value-string. If the composite is not linear, the result is UNDEFINED.

BASIC TRANSFORMATIONS OF TRANSFORMATION CALLS

Transformation calls are themselves data, and they can therefore be operated on by transformations of data. Because they are both complicated data structures, by virtue of being complex strings, and because manipulations on them are frequent, it turns out to be worthwhile to define separate transformations for this purpose.

Conditional selection.

Name: CONDITIONAL-SELECT(c1,r1,c2,r2,c3,r3,...)

Effect: To select a parameter based on which member of a sequence of conditions is satisfied. That is, if c1 is true, select parameter r1, or if c2 is true, pick parameter r2, etc. c1,c2, etc., are references to data elements (they can be either command references

or data references). "If c1 is true", means "if c1 is a reference to a data element having the value-string 'true'".

Exit-value: The selected parameter, a substring presumably interpretable as an identifier. If none of the conditions are true, then the exit-value is an identifier of a cell containing "UNDEFINED".

Comment: The references c1,c2, etc., traditionally are functions having the possible values of "true" and "false". Examples of such functions are "greater than", "less than", "equal to" and their negatives applied to domains of numbers, of Boolean values, and strings. This function corresponds in spirit to McCarthy's conditional expression but not in the details of its definition.

Select a specified parameter.

Name: SELECT(N,A,B,C,...)

Effect: Selects the nth element of the sequence A,B,C.....

Exit-value: A, B, or C, etc.

Comment: This is like the "case expression" introduced by McKeeman in his "An Approach to Computer Language Design". It could be defined in terms of a string scanning function which scans the transformation call and picks the (n + 1)st substring, where parentheses and commas are substring demarcators.

Test existence of a data element.

Name: EXIST(I)

Effect: Tests the existence of a data element referenced by I.

Exit-value: An identifier of a cell containing an identifier of the data element referenced by I, or, if no such data element exists, the value-string UNDEFINED.

Test membership of a data element in a composite.

Name: MEMBER(I,E) or MEMBER(I,E,N)

Effect: Test the existence of data element referenced by E in composite referenced by I. If the parameter N is given, then test the existence of the data element as the nth component, where n is the quantity referenced by N.

Exit-value: Identifier of a cell whose contents are:

1. If the data element referenced by E is a component (or the nth component, if N is specified) of the composite referenced by I, then the cell contents is an identifier of the data element

referenced by E.

2. If the data element referenced by E is not a component of the composite referenced by I, then the cell contents is the value-string UNDEFINED.

Test if two data elements stand in a specific relation.

Name: RELATION(A,R,B)

Effect: To test if two data elements, referenced by A and B respectively, stand in a binary relationship (composite), which composite is, in turn, a component of a larger composite (relation) referenced by R.

Exit-value: If the test yields "true", then the exit value is an identifier of a cell whose contents is an identifier of the relationship in which A and B are components. If the test yields "false", then the exit-value is an identifier of a cell containing the value-string UNDEFINED.

Comment: This function is definable in terms of a search based on the function MEMBER.

Indirect reference.

Name: INDIRECT(I)

Effect: Assumes that the contents of the cell referenced by I is a value-string to be interpreted as an identifier.

Exit-value: An identifier which is the contents of the cell referenced by I.

FUNCTIONS DEFINED ON DOMAINS OF VALUE-STRINGS

The user can define domains as he chooses, and then define an unlimited number of functions on these domains. As discussed earlier under Representation of Functions, any function can be represented either by enumeration (plus a search algorithm) or by some algorithm based on computation rather than search. The representation of a function may be based on other functions. The ultimate definition must be in terms of primitive functions and/or those represented by enumeration.

The user can, by these available function definition methods, define all the functions he wishes. The most common types of functions are those which yield identifiers of simple data elements: in this category belong all the conventional functions such as are illustrated below:

Arithmetic	Logical	String
add	and	concatenate
subtract	or	deconcatenate
multiply	exor	extract substring

Arithmetic (continued)	Logical (continued)	String (continued)
divide	not	string substitution
root	neither	
exponentiate	equivalent	
sin	identical	
cos		

The string functions are somewhat messy to define because a string must be converted to a composite data element and perhaps back again. It is much more convenient to have the string functions given as primitive. One of the more complicated useful functions that can be defined is that of lexicographic sum or difference of two linear ordered composites whose components are from ordered domains.

BASIC TRANSFORMATIONS OF SEQUENCE CONTROL

Transformations of sequence control are those transformations which affect the sequence in which other transformations get executed. The common transformations of this type are: iteration, conditional execution, and chronic execution. Since the sequence controls are themselves data elements, any ordinary transformation which modifies a data element which is part of the sequence control will necessarily affect the sequence of execution. Such transformations of data could thus be classified as transformations of sequence control.

Identifying a control list.

Name: THREAD-IDENT

Effect: Determines the identifier of the control list associated with the thread of control in which the call occurs.

Exit-value: Identifier of the associated control list. If there is no such control list, the result is UNDEFINED.

Initiate control thread.

Name: START(T)

Effect: To start a thread of control at the transformation call whose identifier is T.

Terminate control thread.

Name: STOP

Effect: To stop further execution of the thread in which the transformation STOP occurs.

Branch, or jump.

Name: JUMP(T)

Effect: To stop further execution of the thread in which the transformation

JUMP occurs and to start a thread of control at the transformation whose identifier is T.

Comment: It is obvious that this transformation is simply definable in terms of the combination of STOP and START(T).

Conditional execution.

Name: CONDITIONAL(F,T)

Effect: If the data element referenced by F (which may include parameters) has a value-string of "true", then execute the transformation referenced by T.

Example: Consider the ALGOL statement if A=B then S=T. This would be represented in terms of our conditional as

CONDITIONAL(CALL GR(A,B), "AS(S,T)")

Note that if F was an identifier of the expression CALL GR(A,B) and if T was an identifier of the expression AS(S,T), then the conditional could have been written as CONDITIONAL(F,T)

Chronic execution.

Name: WHENEVER(F,T)

Effect: This is like CONDITIONAL except that the referenced transformation is executed each time the condition is satisfied.

Comment: The execution of a chronic statement at the proper moment depends on having the test made whenever any of the elements involved in the test might change. To implement this is a non-trivial problem whose solution I do not attempt here. It should be obvious that execution of the specified transformation should disturb the condition, otherwise the condition will always be satisfied and the transformation will be executed continually without pause.

Shaw of SDC has proposed an interesting variation on "chronic execution", that of "delayed execution". Delayed execution is similar to chronic, except that execution occurs only once. Delayed execution can be defined in terms of chronic by disabling the transformation call after one execution. Similarly, by use of a counter, the chronic repetition can be limited to some predetermined number of times.

Iteration over members of an explicit set.

Name: ITERATION-OVER-EXPLICIT-SET(S,M,T)

Effect: Executes transformation referenced by T once for each component of explicit set (composite) referenced by S, where the parameter

M in the transformation T references a different component of S on each execution of T. It is important to note that the parameter M in T can be at any depth of nesting, and may occur more than once at the same depth or different depths.

Iteration over members of an implicit set.

This transformation is readily definable in terms of constructing an explicit set from the implicit one and then invoking the transformation ITERATION-OVER-EXPLICIT-SET, which see.

TRANSFORMATIONS OF DOMAIN

Enumerated domains are in the form of ordinary data elements, so that such domains can be created, modified, and destroyed by the ordinary transformation of data. In the case of domains described by algorithm, these algorithms are themselves expressed as data elements. Hence they, too, can be created, modified, and destroyed by means of transformations of data, but at the price of considerable more effort.

BASIC TRANSFORMATION OF PROCESSOR ACTION

These are the transformations which guide the behavior of the processor. Except for explicit special transformations which may be provided, the user is not able to alter or to add to the actions of the processor. This is a logical consequence of the processor being defined outside the language that it is designed to process.

Remote call.

Name: EXECUTE(T)

Effect: Invokes execution of the transformation whose identifier is T.

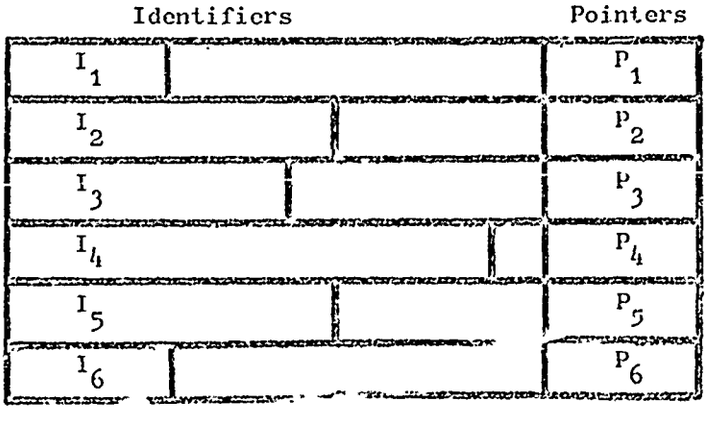
Comment: It is important to note the distinction between this transformation and CALL. Following the word Call is the actual transformation name possibly followed by parameters. Following EXECUTE is an identifier which names a transformation call. Consider the following example. Let K be an identifier of a transformation which has no parameters. Further, let L be an identifier of a data element which contains the string CALL K. Then K can be invoked either by executing CALL K, or by executing EXECUTE L. Both CALL and EXECUTE imply a return to the point of call; this can be made to happen automatically, as is done with the COBOL verb PERFORM. Note that CALL is a special case of EXECUTE which in effect supplies the quotes. CALL K and EXECUTE "K" are equivalent.

CHAPTER 7. REALIZATION OF A PROCESSOR

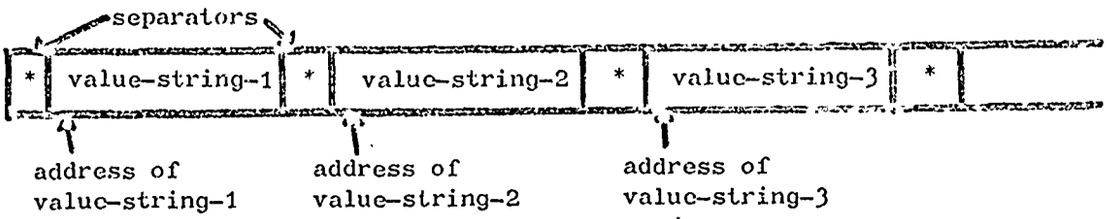
This chapter is concerned with some of the basic considerations of implementing a processor for the system I have been developing. The principal challenge is data storage, for that is the area in which these developments are most unlike current practices. In the discussion that follows, I intend only to show how the problems of realization may be attacked. I am not concerned with efficiency and make no claim that I have illustrated the best way to accomplish certain objectives.

Data storage. The data elements and structures which have been described earlier do not fit readily into any conventional concept of machine storage. My aim here is to describe a storage concept which is at least plausible in relation to what has already been built in hardware.

The first requirement is that there be a place to store and search a set of strings (of arbitrary length up to some fixed limit) which are used as identifiers. That is, they stand in an identifier relationship to other data elements. Such an identifier list could be stored in an associative memory. Each cell of this associative memory would have to be N+P characters long, where N is the maximum length of a pointer, which is a machine address or a quantity analogous to it. Such a pointer points to a memory position containing the initial element of an association list or the initial character of a value-string which is the contents of some simple data element. Both of these latter concepts are about to be explained below. The identifier list as described could be represented pictorially as follows:



The second requirement is to be able to have, as the contents of any simple data element, a value-string of arbitrary length. This implies that we need a memory in which character positions are individually addressable. Value-strings would be referred to by giving the address of the initial character position. The end of each value-string would be marked by some meta-character known only to the processor. Value-string storage could be pictorially represented as follows:



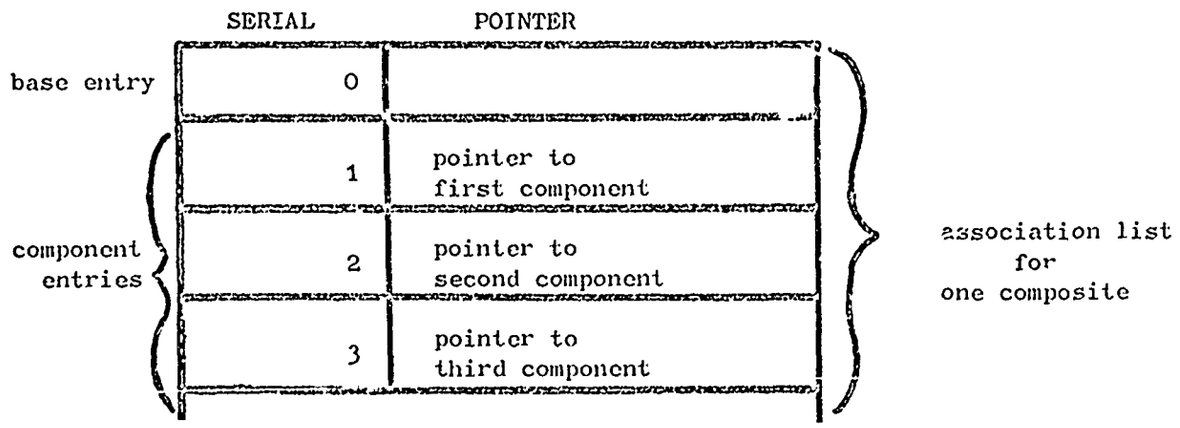
Replacement of a given value-string by a string longer or shorter than the one replaced places on the processor a burden of moving strings and collection of the spaces which become unused as a result of rearrangement. This burden is a nuisance but requires no sophistication for its proper handling.

How value-strings are actually stored internally is the implementor's choice and responsibility. There must be an exact correspondence between external (as seen by the user) and internal (as seen by the processor) representations. The use of approximations resulting from truncation, roundoff, conversion from one number base to another, etc., will lead to trouble.

The third requirement is to be able to associate two or more data elements into a composite data element where the components of this composite can be either a simple or a composite data element. This association, or composite, can be realized by a consecutive set of entries, which I term an "association list", in an associative memory. Each entry in an association list contains a pointer to (that is, a machine address of) a component. If the component is a simple data element, the pointer points to the initial character of a value-string in value-string storage. If the component is itself a composite, the pointer points to another association list, in a way which indicates that the point refers to the association list as a whole.

Each component of a composite, then, has a corresponding entry in an association list which points to that component. Each such component entry can have identifiers related to it; such identifiers are part-identifiers. As just mentioned, there is also a need for a place in an association list which can be pointed to in order to refer to the association list as a whole. This is accomplished by having an extra entry, which we term a "base entry", which is distinguishable from the entries which represent components.

Given a component entry, found by asking the associative memory for those entries which point to a specified data element (that is, to a specific machine address), we must be able to readily locate the base entry associated with that component entry. A simple technique for tying a base entry and the components of a single composite together is to assign serial numbers, 0,1,2,3, etc., to the successive entries. The base entry (of every association list) is thus distinguishable by having a serial number of 0. Each component entry has a serial number, s, which corresponds to its ordinal number. When a component entry is found by the association mechanism, the machine address of the associated base entry can be found by subtracting the serial number of that entry from the machine address of that entry. A pictorial representation of a composite should clarify this discussion:



As with value-string storage, the replacement of an association list by one longer or shorter places on the processor a burden of handling insertions and of collecting unused entries. As mentioned before, this burden is a nuisance but requires no special techniques.

Transformation execution. In earlier discussion I suggested that the task of syntactic analysis of program statements could be rendered

unnecessary by the use of a canonic form for the expression of all transformations. That form is the familiar function notation

T(A,B,C)

where T stands for the name of the desired transformation, and A,B,C stands for the list of actual parameters, if any. Every part of a program which is intended for execution can be put in this form. This form requires no syntactic analysis, since its syntax is known by convention.

The processor's job of handling transformations in canonic form is essentially trivial. Each transformation is a call to a procedure. In the cases of primitive transformations, which are written in machine language, the processor merely looks up the procedure's machine address and branches to it. The primitive procedure is responsible for handling its own parameters.

In the case of transformations defined within the system (that is, are not primitive), the processor looks up the specified transformation, but does not branch to it. Rather, the processor continues to find and trace through the various levels of procedures which may be called, executing any primitive procedures which it encounters in this way. This technique of execution is a standard one, most clearly illustrated in J.W. Carr's "growing machine" discussed in Ostrand, 1967. What is non-standard, and what I think is new here, is the concept that a called procedure is responsible for handling its own parameters rather than having it done by an external device (sometimes called "the interpreter"). This parameter-handling must obviously be done by primitive procedures, otherwise a logical infinite recursion exists. Furthermore, since the parameter-handling is done by explicit transformations, the parameter-handling transformation must be able to reference the call of the transformation at the next-higher level. That is, given a transformation T1, one of the transformations within T1 must be able to pick up and process the parameters in at least the current call to T1. It would be useful if this same parameter-handling transformation could access earlier calls to T1 as well.

Garbage collection. It may happen that a user does not destroy data which he no longer needs. There are two reasons why it is desirable to have garbage collection performed periodically to reclaim this useless

data:

1. To be able to reuse the storage area.
2. To avoid ambiguous situations which might arise from a search yielding some forgotten and therefore unexpected data.

Such garbage collection should be carried out automatically by the processor, without the knowledge of the user. It has already been pointed out that techniques of garbage collection and storage reorganization are well-known and not difficult to implement so the subject will not be elaborated here.

CHAPTER 8. CONCLUSION

Hopefully this work has made a useful contribution to the analysis of programming languages in general and has suggested some useful ways of extending conventional programming language concepts. There is no need to repeat here as a summary what was presented as an overview in Chapter 3. However, Chapter 3 should now have more meaning to the reader.

It hardly needs to be pointed out that this research project has been a paper study. An obvious next step is to test out the ideas in practice. This would involve writing a processor and testing it. After that it should be made available, with a suitable user's manual, to a selected set of users, to let them provide some realistic testing of its usefulness. These users should preferably be ones with non-standard and sometimes ill-structured needs.

In the philosophy developed in this report, every statement in every programming language can be regarded as a call to a transformation. This seems to be a useful concept in analyzing programming languages. It would be worthwhile to see a thorough analysis of current languages based on this approach.

The most significant demand on machine design which arises from the developments of this research project is that much more freedom of storage organization is needed than is provided by conventional machines. The restriction of addressing mechanisms to that of sequentially-numbered storage locations has been a serious one. Accessing such storage has been largely confined to either knowing the storage address or being able to compute it with a simple algorithm (this is the basis for using index registers). When this is not possible, then laborious linear searches, and sometimes binary searches, must be resorted to. Often data not in sequence must first be sorted by a relatively time-consuming technique. With the availability of a parallel-access technique (an associative memory), the labor of searching and sorting is eliminated. A further consequence of the lack of freedom of storage organization has been the retarding of the development of elaborate data structures, because of the difficulty of accessing parts of the structure when it is an irregular one (and therefore difficult to compute the location of the desired part by means of a simple algorithm). Again the utilization of an associative

memory avoids this accessing difficulty and makes attractive the use of complex and irregular data structures. This report suggests a way that large-scale associative memories could be used to provide some of the needed flexibility. One hopes for solutions even "neater" than those provided by associative memories, but there are no other obvious ones within the framework of current technology.

BIBLIOGRAPHY ON PROGRAMMING LANGUAGE, CONCEPTS, AND DESIGN

This bibliography is one man's attempt to list publications relevant to programming language description, concepts, and design. This list should be helpful to researchers concerned with this specific topic.

The reader may wonder why the list lacks any consequential mention of compilers and compiler techniques. This lack reflects a personal conviction that compilers and compiler techniques are not pertinent to the question of how to design better programming languages.

I regret to say that I have not read all the material cited here. Some of it is entered because it intuitively satisfied my criteria of probable relevance, based usually on a glance at the document but based sometimes only upon consideration of the title.

A brief annotation of each citation would be helpful. To provide such annotations is a non-trivial task, however, which time did not allow. Hopefully on the next revision of this bibliography those annotations can be provided, whether by me or by someone else.

Abrahams, Paul., et al. The LISP2 programming language and system. Doc. no. TM-3163. Santa Monica, Calif.,: SDC, 1966.

Albani, E., Ceccato, S., and Maretti, E. Classifications, rules, and code of an operational grammar for machine translation. IN: Kent, A., ed. Information retrieval and machine translation, part 2. New York: Interscience, 1961; 693-753.

Almendinger, V.V. SPAN: A system for urban data management. IN: Computer Yearbook and Directory. Detroit: American Data Processing, Inc.; 175-182.

Alzati, E.C. An experimental programming language for teaching symbolic manipulation. Moore School of Electrical Engineering, University of Pennsylvania, May 1966. AD 800 050.

Amarel, Saul. An approach to theory formation. IN: von Foerster, H., ed. Principles of self organizing systems. Pergamon, 1962.

Amarel, Saul. On the automatic formation of a computer program which represents a theory. IN: Yovits, M.C., et al. Self-organizing systems—1962. Washington, D.C.: Spartan Books, 1962.

Amarel, Saul. On the mechanization of creative processes. IEEE Spectrum 3,4 (April 1966), 112-114.

American Institutes for Research. (Self-instructional Course) Mod II Query Language. Doc. no. ESD-TR-66-513. Bedford, Mass.: Decision Sciences Laboratory, Electronic Systems Division, Air Force Systems Command, 1966. AD 808 212 through AD 805 240.

Ammerman, A.B., Deisen, L.R., and Thombs, H.W. DISPLAYTRAN—a graphical display oriented conversational FORTRAN facility for an IBM 360/40 computer. Dahlgren, Va.: U.S. Naval Weapons Lab., 1967. AD 656 583.

Armenti, A.W., Schafer, B.J., and Winett, J.M. A data processing formalism. TR 283. Massachusetts Institute of Technology, Lincoln Laboratory, 1962.

Bachman, C.W., Dodd, G.G., Durand, G.C., Helgeson, W., McKenzie, G.E., Metaxides, T.A., Olle, T.W., et al. COBOL extensions to handle data bases (report to Codasyl COBOL Committee, prepared by Data Base Task Group), January, 1968.

Bagley, P.R. A file structure for a library data processing application. Doc. no. W-4657. Bedford, Mass.: The MITRE Corp., 1962.

Bagley, P.R. Improving the art of programming (Project 501). Doc. no. W-4554. Bedford, Mass.: The MITRE Corp., 1962.

Bagley, P.R. Improving problem-oriented language by stratifying it. Computer Journal, Oct. 1961; 217-221.

Bagley, P.R. An introduction to ALGOL and its processors. Philadelphia, Pa.: Moore School of Electrical Engineering, University of Pennsylvania, 1965.

Bagley, P.R. Principles and problems of a universal computer-oriented language. Computer Journal, Jan. 1963; 305-312.

Bailey, M.L. Toward a disciplined data systems language. IN: Computer Yearbook and Directory. Detroit: American Data Processing, Inc.; 107-116.

Bakker, J.W. de. Formal definition of programming languages with an application to the definition of ALGOL 60. Amsterdam: Mathematisch Centrum, 1967.

Ball, J.R., et al. On the use of the SOLOMON parallel-processing computer. Proc. FJCC, 1962; 137-146.

Ball, William E., and Berns, Robert I. AUTOMAST: Automatic mathematical analysis and symbolic translation. Comm. ACM, 9 (1966); 626-633.

Balzer, R.M. Dataless programming. Doc. no. RM-5290-ARPA. Santa Monica, Calif.: RAND Corp., July 1967. AD 656 449.

Bandat, K., ed. Tentative steps towards a formal definition of semantics of PL/I. Doc. no. TR 25.056. Vienna: IBM Laboratory, 1965.

Banerji, R.B. The description list of concepts. Comm. ACM, 5 (1962); 426-432.

Banerji, R.B. A formal model for concept description and manipulation. Moscow: Proceedings of the 18th International Congress of Psychology, 1966.

Banerji, R.B. A language for pattern recognition. Journal of the Pattern Recognition Society (to be published).

Banerji, R.B. A language for the description of concepts. IN: Yearbook of the Sciety for General Systems Research, g. 1964; 135-141.

Banerji, R.B. Towards a formal language for describing object classes. IN: Information system sciences, proceedings of the 2nd congress (1964) Washington, D.C.: Spartan Books, 1965; 451-457.

Barbieri, R., and Morrissey, J. Computer compiler organization studies. Doc. no. AFCRL-67-0483. New York: John Morrissey Associates, 1967. AD 658 196.

Barnes, R.F. Language problems posed by heavily structured data. Comm. ACM, 5 (1962); 28-34.

Barnett, M.P. Continued operation notation for symbol manipulation and array processing. Comm. ACM, 6(1963); 467-472.

Barnett, M.P., Gerard, J.M., and Sambles, A.W. Comments on "A continued operation notation." Comm. ACM, 7(1964); 150-152.

Barron, D.W., et al. The main features of CPL. Computer Journal, 6(1963); 134-142.

Barton, R.S. Notes for lectures on program structures, programming languages, and machine organization. University of Michigan Summer Engineering Conference on Advanced Programming, Machine and Program Organization, 1964.

Basacher, R., and Saaty, T. Finite graphs and networks. New York: McGraw-Hill, 1965.

Basu, S.K. On computation in programming languages. Technical Report #1. Bombay: Tata Institute of Fundamental Research, 1966.

Bauer, H.R., et. al. ALGOL W. Stanford Computer Science Dept., 1968.

Baum, C. ed. Proceedings of the second symposium on computer-centered data base systems. Doc. no. TM-2624/100/00. System Development Corporation, 1965.

Bennett, Richard K. A base for the definition of computer languages. Lexington, Mass.: Signatron, Inc., Oct. 1967.

Bennett, Richard K. BUILD: a base for uniform language definition: a user's manual for computer language designers and systems programmers. Lexington, Mass.: Signatron, Inc., June 1968.

Berge, Claude. The theory of graphs. New York: Wiley, 1962.

Berkeley, E.C., and Bobrow, D.G., eds. The programming language LISP: its operation and applications. Cambridge, Mass.: Information International, 1964. AD 603 482.

Bernstein, A.J., and Johnston, J.B. Implementation of a parallel processing language. Report no. 67-C-080. Schenectady, N.Y.: General Electric, Research and Development Center, March 1967; 17 pp.

Biggins, S.A. APT—a new system. IN: Data Processing, vol. 8 no. 6(Nov-Dec 1966); 309-313.

Blackwell, Frederick W. An on-line symbol manipulation system. Proc. ACM, 1967; 203-209.

Bleier, R.E. User's manual for GOR: A general purpose display system (GPDS) program for structuring LUCID-produced but unorganized data bases. Doc. no. TM-2302. Santa Monica, Calif.: SDC, 1965.

Bleier, R.E. Treating hierarchical data structures in the SDC time-shared data management system (TDMS) Proc. ACM, 1967; 41-49.

Bleier, R.E. User's manual for QUIZ: A data based query system. Doc. no. TM-2429/000/00. Santa Monica, Calif.: SDC, 1965.

Bobrow, Daniel G., and Raphael, Bertram. A comparison of list-processing computer languages. Comm. ACM, 7(1964); 231-240.

Bobrow, D.G. et al. Format-directed list processing in LISP. Doc. no. AFCRL-66-302. Cambridge, Mass.: Bolt, Beranek and Newman, 1966. AD 633 242.

Bobrow, D.G. and Weizenbaum, J. List processing and extension of language facility by embedding. IEEE Trans. EC-13 (1964); 395-400.

Bobrow, D.G. Storage management in LISP. Doc. no. AFCRL 66-426. Cambridge, Mass.: Bolt, Beranek, and Newman, 1966. AD 636 049.

Bobrow, D.G., ed. Symbol manipulation languages and techniques. Proc. IFIP Working Congress on Symbol Manipulation Languages. Pisa, 1967. Amsterdam: North Holland Publishing Co., 1968.

Bohm, Corrado, and Jocopini, Giuseppe. Flow diagrams, Turing machines and languages with only two formation rules. Comm. ACM, 9(1966); 366-371.

Bolliet, L. Compiler writing techniques. Grenoble, France: Institute de Mathematiques Appliques, Sept. 1966; 216 pp.

Bond; Elaine, et al. FORMAC—an experimental formula manipulation compiler. Doc. no. TR 00.1192-1. Poughkeepsie, N.Y.: IBM DSD, 1965.

Book, E. and Schorre, D.V. Higher-level machine-oriented languages as an alternative to assembly languages. TM-3086/001/00. SDC, August 1966.

Bosak, R. CLEAR, an experimental data base management system. Doc. no. TM-2673/000/00. Santa Monica, Calif.: SDC, 1965.

Bosak, R. A proposed file processing language. Doc. no. TM-3392/000/00. Santa Monica, Calif.: SDC, 1967.

Brach, J.R. A problem-oriented language for project management. M.I.T Masters thesis, 1965.

Braffort, P., and Hirschberg, D., eds. Computer programming and formal systems. Amsterdam: North-Holland, 1963.

Brian, W.J. A parts breakdown technique using list structures. Comm. ACM, 7(1964); 362-365.

Broise, P. Tree-structure processing by the method of "Label Sequences" in ALGOL. IN: Proceedings, 4th Congress of Computation and Information Processing, Versailles: 1964. Paris: AFIRO, 1965; 79-86.

Brooker, R.A., and Rohl, J.S. Simply partitioned data structures: the compiler-compiler re-examined. IN: Collins and Michie, eds. Machine intelligence 1. American Elsevier Publishing Co, 1967; 229-239.

Brouse, R.A. The data sequenced computer. Doc. no. FN-4060. Santa Monica, Calif.: SDC, 1960.

Brown, George W. A new concept in programming. IN: Greenberger, M., ed. Management and the computer of the future. New York: Wiley, 1962; 251.

Brown, W.S. et al. The ALPAK system for non-numerical algebra on a digital computer. Bell System Technical Journal, 42(1963); 2081-2119. 43(1964); 785-804, 1547-1562.

Bryant, J.H., and Semple, P., Jr. GIS and file management. ACM Proc. 1966; 97-107.

Brzozowski, J.A. and McCluskey, E.J. Jr. Signal flow graph techniques for sequential circuit state diagrams. IEEE Trans. E.C. 1963; 67-76.

Buckles, Glenn A. and Carpenter, Stewart B. Computer-centered data systems. Doc. no. ESD-TR-66-499. Bedford, Mass.: Electronic Systems Division, Air Force Systems Command, 1966. AD 641 267.

Burge, W.H. The evaluation, classification, and interpretation of expressions. Proc. ACM 19th National Meeting, 1964, Paper A1.4; 22 pp.

Burge, W.H. Interpretation, stacks, and evaluation. IN: Wegner, P., ed. Introduction to system programming. London: Academic Press, 1964.

Burge, W.H. A reprogramming machine. Comm. ACM, 9(1966); 60-66.

Burge, W.H. Sorting, trees, and measures of order. Information and control, 1(1958); 181_197.

Burge, W.H. The structure of the programming language PL/I. New York: Univac Division, Sperry Rand Corp., 1965.

Burington, Richard S. The problem of formulating a problem. Proc. American Philosophical Society, 104(1960); 429-443.

Burington, Richard S. The structure of science: the problem of formulating a problem. R-14 Report no. 24. Washington, D.C.: Navy Dept., 1964. AD 644 733.

Burkhardt, W.H. Metalanguage and syntax specification. Comm. ACM, 8(1965); 304-305.

Burkhardt, W.H. PL/I: an evaluation. Datamation, Nov. 1966; 31-34, 37, 39.

Burkhardt, W.H. Universal programming language and processors: a brief survey and new concepts. Proc. FJCC, 1965; 1-21.

Burley, H.T. A programming language for linear algebra. Computer Journal, May 1967; 67-73.

Burrows, J.H. Automated data management (ADAM). IN: Proceedings of the Symposium on Development and Management of a Computer-Centered Data Base. BRT-41. Santa Monica, Calif.: SDC, 1964; 63-86.

Burrows, J.H. Program structure for military real-time systems. Doc. no. SR-122. Beford Mass.: The MITRE Corp., 1964. Also IN: Proceedings of the Symposium on Computer Programming for Military Systems, the Hague, 28 September - 2 October 1964. The Hague, Netherlands: SHAPE Technical Centre, 1965, vol. 1; 195-224.

Buxton, J.N. and Laski, J.G. Control and simulation language. Computer Journal, 5(1962); 194-199.

Buxton, J.N. Writing simulations in CSL. Computer Journal 9(1966); 137-143.

Caracciolo di Forino, A. Linguistic problems in programming theory. IN: Information Processing 1965. vol. 1. Washington, D.C.: Spartan Books, 1965; 223-228.

Carey, L.J. and Kroger, A.E. Specification of SPL Space Programming Language. Doc. no. TM-3719/000/00. Santa Monica, Calif.: SDC, Sept. 1967. AD 661 981.

Carr, J.W. III. The future of programming and programmers. Computer Bulletin, June 1964; 9-12.

Carr, J.W. III, et al. LIST processing research techniques. Doc. no. ECOM-02377-1. U.S. Army Electronics Command, 1966. AD 645 445.

Carr, J.W. III. and Gray, H.J. LIST processing research techniques. Doc. no. ECOM-02377-2. U.S. Army Electronics Command, 1967. AD 652 724.

Chamberlain, E. et al. The role of locational control in an informational system. Dept. of Housing and Urban Development, Sept. 1967. PB 177 623.

Chandler, A.R. AGIL II: a general input language for on-line information control. ESD-TR-66-308. Doc. no. MTP-12. Bedford, Mass.: The MITRE Corp., 1966. 2 vols. AD 489 421 and 489 422.

Chapin, N. SYLATINS: a way out of the systems maze? Automatic Control, April 1960; 37-43.

Char, B.F. et al. Final report-a joint AFLC/ESD/MITRE advanced data management (ADAM) experiment. Doc. no. MTR-285. ESD-TR-66. Bedford, Mass.: The MITRE Corp., 1967. AD 648 226.

Cheatham, T.E., Jr. Data description in the CL-II programming system. ACM National Conference, Digest of Technical Papers, 1962; 30.

Cheatham, T.E., Jr. An introduction to the CL-II programming system. Doc. no. CA-63-7-SD. Wakefield Mass.: Computer Associates, 1963.

Cheatham, T.E., Jr. The introduction of definitional facilities into higher-level programming languages. Proc. FJCC, 1966.

Cheatham, T.E. Jr. The specification and syntactic analysis of two dimensional mathematical input. Doc. no. CA-6611-0812. Wakefield, Mass.: Massachusetts Computer Associates, 1966; 36 pp.

Cheydleur, Benjamin F. Dimensioning in an associative memory. IN: Howerton, P.W. ed. Vistas in Information Handling, vol 1. Washington, D.C.: Spartan, 1963; chapter 3.

Chien, T.R. and Preparata, F.P. Topological structures of information retrieval systems. Doc. no. R-325. Coordinated Science Laboratory, University of Illinois, 1966. AD 642 501.

Chladek, H., Kudielka, V., and Neuhold, E. Syntax of PL/I. Doc. no. TR 25.058. Vienna: IBM Laboratory, 1965.

Christensen, C. AMBIT: a programming language for symbol manipulation. Doc. no. AFCRL-64-909. Wakefield, Mass.: Computer Associates, 1964. AD 608 894.

Christensen, C. An example of the manipulation of directed graphs in the AMBIT/G programming language. Report no. CA-6711-1511. Wakefield, Mass.: Massachusetts Computer Associates.

Christensen, C. Examples of symbol manipulation in the AMBIT programming language. ACM Proc., 1965; 230-247.

Christensen, C. On the implementation of AMBIT, a language for symbol manipulation. Comm. ACM, 9(1966); 570-573.

Christensen, C. and Mitchell, Robert. Reference manual for the NICOL 2 programming language. Doc. no. CA-6701-2611. Wakefield, Mass.: Massachusetts Computer Associates, Inc., 1967.

Chu, Y. An ALGOL-like computer-design language. Comm. ACM, 8(1965); 607-615.

Church, Alonzo. The calculi of lambda conversion. Princeton, J.J.: Princeton University Press, 1940.

Clementson, A.T. Extended control and simulation language. Computer Journal, 9(1966); 215-220.

Clippinger, R.F. FACT. Computer Journal, 5(1962); 112-119.

Clippinger, R.F. FACT - a business compiler; description and comparison with COBOL and Commercial Translator. IN: Goodman, R., ed. Annual Review in Automatic Programming, vol. 2. 1961.

CODASYL. COBOL 65. Washington, D.C.: GPO, 1965.

CODASYL Development Committee, Language Structure Group. An information algebra, Phase I report. Comm. ACM, 5(1962); 190-204.

CODASYL Systems Group, Data Description and Transformation Logic Task Forces. DETAB-X, Preliminary specifications for a decision table structured language. Sept. 1962. (Available from ACM.)

Cohen, Jacques. Definition of LISP procedures in ALGOL; example of utilization. Revue francaise de traitment de l'information; 1965.

Cohen, Kenneth, and Wegstein, J.H. AXLE: an axiomatic language for string transformations. Comm. ACM, 8(1965); 657-661.

Collins, N.L., and Michie, D. Machine Intelligence I. New York: American Elsevier, 1967.

Collins, G.E. PM, a system for polynomial manipulation. Comm. ACM, 9(1966); 578-589.

Compton, J.B. and Moody, D.E. Flowgraphing executive problems. IEEE Trans. on Engineering Management, vol. EM-12 (1965); 143-149.

Computer Associates. Advanced programming developments: a survey. Doc. no. ESD-TDR-65-171. Bedford, Mass.: USAF, Air Force Systems Command, Electronic Systems Division, Directorate of Computers, 1965.

Connors, T.L. ADAM--a generalized data manag ment system. Proc. FJCC, 1966; 193-203.

Connors, T.L. A brief view of ADAM. Document W-7047. Bedford, Mass.: The MITRE Corp., 1964.

Constantinescu, Paul. The classification of a set of elements with respect to a set of properties. Computer Journal, 8(1966); 352-357.

Conway, Melvin E. A multiprocessor system design. Proc. FJCC, 1963.

Cooper, D.C. Computer programs and graph transformations. Pittsburgh: Carnegie Institute of Technology, 1966.

Cozier, W.A. and Dennis, W.C. QUUP user's manual. Doc. no. TM-2711/000/00. Santa Monica, Calif.: SDC, 1965.

Craig, J.A. et al. DEACON: direct English access and control. Proc. FJCC, 1966; 365-380.

Curry, H.B. and Feys, R. Combination logic, vol. 1. Amsterdam: North Holland, 1958; 433 pp.

Davis, Martin, and Putnam, Henry. A computing procedure for quantification theory. Jour. ACM, 7(1960); 201-215.

Davis, Ruth M. Programming language processors. IN: Alt, F., ed. Advances in Computers, vol. 7. New York: Academic Press, 1966; 117-180.

de Bakker, J.W. Formal definition of algorithmic languages with an application to the definition of ALGOL 60. Doc. no. MR 74. Amsterdam: Mathematical Centre, May 1965.

Dennis, Jack, and Van Horn, Earl C. Programming semantics for multiprogrammed computations. Doc. no. MAC-TR-23. Cambridge, Mass.: M.I.T. Project MAC, 1965. AD 627 537.

Deuel, P. On a storage mapping function for data structures. Comm. ACM, 9(1966); 344-347.

Dijkstra, E.W. A simple mechanism modelling some features of ALGOL 60. ALGOL Bulletin #16(1964); 14-23.

d'Imperio, Mary. Data structures and their representation in storage: Part I. NSA Technical Journal, 9(1964); 59-81.

d'Imperio, Mary. Data structures and their representation in storage: Part 2. NSA Technical Journal, 9(1964); 7-54.

Dimsdale, B., and Markowitz, H.M. A description of the SIMSCRIPT language. IBM Systems Journal, 3(1964); 57-67.

Dittberner, D.L. Implementation of communications-based EDP information systems (workshop summary). IEEE Trans. on Electronic Computers, 1966; 271-272.

Douglas, M., Farber, D., Rosin, R., Shellans, S. Brief guide to PL/I for programmers experienced in FORTRAN and other algebraic languages. SHARE Secretary Distribution. C4415, August 1966.

Dreyfus, Hubert L. Alchemy and artificial intelligence. Document P-3244. Santa Monica, Calif.: RAND Corp., 1965. AD 625 719.

Dueker, Kenneth, J. Spatial data systems: organization of spatial data. Northwestern University, 1966. AD 652 005.

Dueker, Kenneth, J. Spatial data systems: special topics. Northwestern University, 1966. AD 652 007.

Duly, J.R. DDL—a DIGITAL SYSTEM DESIGN LANGUAGE. University of Wisconsin Ph.D. thesis. May, 1967; 213 pp.

Dzubak, B.J., and Warburton, C.R. The organization of structured files. Comm. ACM, 8(1965); 446-452.

Edelstein, L.A. "Picture Logic" for "Bacchus," a fourth-generation computer. Computer Journal 6(1963); 144-153.

Efron, R., and Gordon, G. A general purpose digital simulator and examples of its application, Part 1, description of the simulator. IBM Systems Journal, 3(1964); 22-34.

Elmaghraby, S.E. An algebra for the analysis of generalized activity networks. Managment Science, 3(1964); 494-514.

Engeli, M. Achievements and problems in formula manipulation. Proc. IFIP Congress, Edinburgh, 1968.

Estrin, G. Organization of computer systems—the fixed plus the variable structure computer. Proc. WJCC; 33-40.

Evans, D., and Van Dam, A. Data structure programming system. Proc. IFIP Congress, Edinburgh, 1968.

Evans, David S. Data structures for man-machine communication in network analysis. Proc. of the First Annual Princeton Conference on Information Sciences and Systems, 1967; 306-309.

Evans, D.S. and Katzenelson, J. Data structure and man-machine communication for network problems. Proc. IEEE, 55(1967); 1135-1144.

Evans, O.Y. General information manual: advanced analysis method for integrated electronic data processing. Doc. no. F20-8047. White Plains, N.Y.: IBM Corp., 1960.

Falkoff, A.D. Algorithms for parallel-search memories. Jour. ACM, 9(1962); 488-511.

Farber, D.J., Griswold, R.E., and Polonsky, I.P. SNOBOL, a string manipulation language. Jour. ACM, 11(1964); 21-30.

Farber, D.J. et al. The SNOBOL3 programming language. Bell System Technical Jour., 1966; 895-944.

Feder, Jerome. The linguistic approach to pattern analysis: a literature survey. Bronx, N.Y.: New York University, 1966. AD 637 497.

Feder, Jerome. Linguistic specification and analysis of classes of patterns. Doc. no. TR 400-147. Bronx, N.Y.: Laboratory for Electroscience Research, New York University, 1966. AD 651 606.

Feingold, S.L. and Frye, C.H. User's guide to PLANIT, Programming Language for Interactive Teaching, Doc. TM-3055. Santa Monica, Calif.: SDC, 1966.

Feldman, J.A. Aspects of associative processing. Technical Note 1965-13. Lexington, Mass.: MIT Lincoln Laboratory, 1965.

Feldman, J.A. A formal semantics for computer oriented languages. Pittsburgh, Pa.: Carnegie Institute of Technology, 1964.

Fenves, S.J. Problem-oriented languages for man-machine communication in engineering. IN: Proc., IBM Scient. Comp. Symp., Man-Machine Communications. New York, 1965. IBM report no. 320-1941-0, 1966; 43-56.

Fisher, D.L. Data, documentation and decision tables. Comm ACM,9(1966);26-31.

Floyd, R.W. A descriptive language for symbol manipulation. Jour. ACM, 8(1961); 579-584.

Floyd, R.W. The syntax of programming languages: a survey. IEEE Trans. EC-13 (1964); 346-353.

Fox, L., ed. Advances in programming and non-numerical computation. New York:Pergamon, 1966.

Franks, E.W. A data management system for time-shared file-processing using a cross-index file and self-defining entries. Doc. No. SP-2248. Santa Monica, Calif.: SDC, 1966.

Franks, E.W. General purpose data management systems for administrative and for socio-economic planning and analysis: SPAN and IDMS compared. SDC report no. SP-2652, 1966.

Franks, E.W. LUCID (Language Used to Communicate Information-Systems Design) IN: Proceedings of the Symposium on Development and Management of a Computer-centered Data Base, June, 1963. Doc. no. BRT-41. Santa Monica, Calif.: SDC, 1964; 87-96.

Frye, W.H. The interrelationship graph as a technique for computer model development. Doc. no. NEL-1322. San Diego, Calif.: Naval Electonics Lab., 1965. AD 482 626.

Garofano, Ralph. A graph theoretic analysis of citation index structures. Masters thesis. Philadelphia, Pa.: Drexel Institute of Technology, 1965.

Garwick, J. et al. GPL. Technical Report TER-D-5. Control Data Corporation.

Gaskill, R.A. A versatile problem-oriented language for engineers. IEEE Trans. EC-13(1964); 415-421.

Gilbert, P., Gunn, D.M., and Schager, C.L. Automatic programming techniques. Report no. TR-66-54. Teledyne System Corp., Rome Air Development Center, 1966.

Gill, A. Analysis of linear sequential circuits by confluence sets. Doc. no. 64-30. Berkeley, Calif.: University of California, 1964. AD 607 476.

Gill, Stanley. The changing basis of programming. IN: Information Processing 1965. Washington D.C.: Spartan Books, 1965; vol. 1, 201-206.

Gilmore, John T., Jr. Research on advanced dynamic attribute extraction techniques. Cambridge, Mass.: Charles W. Adams Associates, 1965. Doc. no. AFCRL-65-736. AD 625 181.

Gorn, S. On the logical design of formal mixed languages. IN: Gorn, S. et al. Common programming language test, Part I, Doc. no. AD59UR1, vol.4, Task D Final Report, Philadelphia, Pa.: Moore School of Electrical Engineering, University of Pennsylvania,1959; 41-160. Also available as AD 236 897. CONTINUED IN: Gorn, S., et al. Common programming language task, Part 1, Doc. no. AD6OUR1, vol.4, Task D, Final Report. Philadelphia, Pa.: Moore School of Electrical Engineering,University of Penna. 1960; 1-101. Also available as AD 248 110.

Gorn, S. The computer and information scie.ces: a new basic discipline. SIAM Review, 5(1963); 150-155.

Gorn,S. The treatment of ambiguity and paradox in mechanical languages. IN: Recursive Function Theory, Proceedings of Symposia in Pure Mathematics, vol. 5. Providence, R.I.: American Mathematical Society, 1962.

Gorn, S. Semiotic relationships in ambiguously stratified language systems. Paper presented at the International Colloquium for Algebraic Linguistics and Automata Theory. The Hebrew University of Jerusalem, 1964.

Gould, M.J. and Logemann, G.W. ALMA: Alphameric language for music analysis. Institute for Computer Research in the Humanities, New York University, 1966.

Grant, E.E. The LUCID users' manual.Doc.no.TM-2354/001/00. SDC, 1965.

Green, Robert S. Analysis of small associative memories for data storage and retrieval systems. Doc. no.RADC-TR-65-397. Rome Air Dev. Ctr.,AD 489 661.

Greenberger, C.B. The automatic design of a data processing system. IN: Information Processing,1965. Washington,D.C: Spartan Books,1965, vol. 1, 277-282.

Greenberger, Martin. A new methodology for computer simulation. MAC-TR-13. Cambridge, Mass.: Massachusetts Institute of Technology, 1964.

Grindley, C.B.B. Systematics—a non programming language for designing and specifying commerical systems for computers. The Computer Journal, 9 (1966); 124-128.

Griswold, R.E. Poage, J.F., and Polonsky, I.P. The SNOBOL4 programming language. Doc. no. S4D9. Holmdel, N.J.: Bell Telephone Laboratories, 1968.

Gumin, H.W. The influence of programming languages on the organization of digital computers. IN: Information Processing 1962. Amsterdam: North-Holland, 1963; 566-567.

Guzman, Adolofo, and McIntosh, Harold V. CONVERT. Comm.ACM, 9(1966); 604-615.

Haimes, E.C. The TREET list-processing language. Doc. no. SR-133. Bedford, Mass.: The MITRE Corp., 1965.

Halpern, M.I. Standardize the system, not the language. IN: Proc. of the 1st Spacebor..e Computer Software Workshop, 1966; 211-221.

Hamblin, C.L. Computer languages. The Australian Journal of Science, (1957); 135-139.

Harary, F., Norman, R., Cartwright, D. Structural models: an introduction to the theory of directed graphs. New York: Wiley, 1965.

Harris, E.H. STRIGOL: a string oriented language. CFSTI Rpt. no. N67-40067. New Mexico State University, 1967.

Haynam, G.E. An extended ALGOL based language. Proc. ACM 20th National Conference, 1965; 449-454.

Heller, J., and Logemann, G.W. PL/I: programming language for humanities research.IN: Computers and the Humanities, Nov. 1966.

Henderson, P.B., Jr. A theory of data systems for economic decisions. Ph.D. thesis, M.I.T., June, 1960.

Henderson, V.D. and Smith, G.L. Efficiency considerations in problem-oriented processor design. IN: Proc. 1st Spaceborne Computer Software Workshop, 1966; 20-22.

Herscovitch, H., and Schneider, T.H. GPSS III - an expanded general purpose simulator. IBM Systems Journal, 4(1965).

Higman, Bryan. A comparative study of programming languages. New York: American Elsevier, 1967.

Hoare, C.A.R. Further thoughts on record handling. ALGOL Bulletin #23 (May 1966); 5-12.

Hoare, C.A.R. The limitations on languages. IN: Computer Weekly, no. 98; London; August, 1968; 6.

Hoare, C.A.R. and Wirth, N. Proposal for a successor to ALGOL 60. Comm. ACM, 1966.

Hoare, C.A.R. Record Handling. ALGOL Bulletin 21 (1965); 39-69.

Hoffman, Samuel A. Data structures that generalize rectangular arrays. Proc. SJCC, (1962); 325-333.

Hoffman, H.J. A proposal for input-output handling in ALGOL 60. ALGOL Bulletin #20. (1965); 31-45.

Holden, Alistair, D.C., and Johnson, David L. The use of imbedded patterns and canonical forms in a self-improving problem solver. Proc. ACM, (1967); 211-219.

Holmes, W.N. Some remarks on PL/I. PL/I Bulletin no. 2. Aug. 1966; 22-24.

Holt, A.W. Description of descriptions. Princeton, N.J.: Applied Data Research, c. 1964.

Holt, Anatol, W. General Purpose programming systems. Doc. no. TR 16. Univac Applications Research Center, 1957. Comm. ACM. (1958); 7-9.

Holt, Anatol, W., et al. Information system theory project. vol. 1. Memtheory: a mathematical method for the description and analysis of discrete, finite information systems. Doc. no. RADC-TR-65-377. Griffiss Air Force Base, N.Y.: Rome Air Development Center, Nov. 1965. AD 626 819.

Holt, A.W. Some theorizing on memory structure and information retrieval. Princeton, N.J.: Applied Data Research, Oct. 1963.

Hopcraft, J.E. and Ullman, J.D. Formal languages and their relation to automata. Addison-Wesley. Scheduled for publication early 1969.

Hormann, A.M. Introduction to ROVER, an information processor. Doc. no. FN-3487. Santa Monica, Calif.: SDC, 1960.

Horrigan, T.J. Development of techniques for prediction of system effectiveness. Doc. no. RADC-TDR-63-407. Morton Grove, Ill.: Cook Electric Co., 1964.

Iliffe, J.K. The elements of the GENIE system, programming memorandum #4. Houston, Texas: Rice Institute Computer Project, 1960.

Iliffe, J.K. The use of the GENIE system in numerical calculation. IN: Goodman, R., ed. Annual Review in Automatic Programming, vol. 2. Oxford; Pergamon Press, 1961.

Ingerman, Peter Z. A syntax-oriented translator. New York: Academic Press, 1966.

IBM Corp. FORMAC manual. Hawthorne, N.Y.: IBM Corp., Program Information Dept.

IBM Corp. Generalized information system: application description. Doc. no. E20-0179. White Plains, N.Y.: IBM, 1965.

IBM Corp. IBM operating system/360. PL/I: language specifications. Doc. no. C28-6571. White Plains, N.Y.: IBM, 1965.

IBM Corp. Introduction to the compile-time facilities of PL/I. Report no. C20-1689. IBM Corp., 1968.

IBM Corp. System 360 generalized information system (basic), application description. Report no. H20-0521. IBM Corp., 1968.

Iturriaga, R. Contributions to mechanical mathematics. AFOSR-67-2400. AD 660 127.

Iturriaga, Renato et al. The implementation of formula ALGOL in FSL. Carnegie Institute of Technology, Center for the Study of Information Processing. 1966.

Iverson, K.E. Formalism in programming languages. Comm ACM, 7(1964); 80-88.

Iverson, K.E. A programming language. New York: Wiley, 1962.

Johns Hopkins University Applied Physics Laboratory. APT, Common Computer Language, 1958.

Johnson, L.R. On operand structure, representation, storage and search. Doc. no. RC-603. Yorktown Heights, N.Y.: IBM Corp., T.J. Watson Research Center, 1961.

Johnson, T.E. Sketchpad III: a computer program for drawing in three dimensions. Proc. SJCC, 1963.

Kameny, S.L. LISP 1.5 reference manual for Q-32. Doc. no. TM-2337/101/00. Santa Monica, Calif.: SDC, 1965. AD 622 018.

Kameny, S.L. (SDC),Hawkinson L. (III). LISP II project: LISP II intermediate language. Doc. no. TM-2710/220/00. Santa Monica, Calif.: SDC/III, 1965.

Kameny, S. Three languages in solution of a problem: "man or boy?" in ALGOL 60, LISP2, and LISP1.5. SICPLAN Notices, Oct. 1966; 7-10.

Kaplow, R., Brackett, J., Strong, S. Man-machine communication in on-line mathematical analysis. Project MAC. Proc. FJCC 1966; 465-477.

Karp, R.M., and Miller, R.E. Parallel program schemata. IBM Research Rep. RC 2053. Yorktown Heights, N.Y.: April, 1968.

Karr, H.W., Kleine, H., and Markowitz, H.M. SIMSCRIPT 1.5. Santa Monica, Calif.: California Analysis Center, 1966.

Katz, J.H. and McGee, W.C. An experiment in non-procedural programming. Proc., 1963; 1-13.

Kavanagh, T.F. TABSOL: a fundamental concept for systems-oriented languages. Proc. EJCC., 1960; 117-136.

Kay, Martin, and Ziehe, Theodore. The catalog: a flexible data structure for magnetic tape. Proc. FJCC, 1965; 283-291.

Kellogg, Charles H. An approach to the on-line interrogation of structured files of facts using natural language. Doc. no. SP-2431/000/00. Santa Monica, Calif.: SDC, 1966.

Kim, C.K., et al. A new approach to computer command structures. Doc. no. RADC-TDR-63-135. Dayton, Ohio: Systems Research Labs., 1964. AD 607 363.

King, W.F. State-logic relations in an iterative structure for autonomous sequential machine. Doc. no. AFCRL-65-439. Bedford, Mass.: AFCRL, 1965. AD 619 806.

Kirsch, R.A. Computer interpretation of English text and picture patterns. IEEE Trans. EC-13(1964); 363-376.

Kiviat, P.J. Development of new digital simulation languages. Doc. no. P-3348. Santa Monica, Calif.: RAND Corp., 1966.

Kiviat, P.J. Introduction to the SIMSCRIPT II programming language. Doc. no. P-3314. Santa Monica, Calif.: RAND Corp., 1966.

Klerer, M., and May, J. Two-dimensional programming. Proc. AFIPS Fall Joint Computer Conference, 1965; pt. 1; 63-73.

Klerer, M., and May, J. A user oriented programming language. Computer Journal., 8(1965); 103-109.

Knowlton, Kenneth C. A programmer's description of L^6. Comm. ACM, 9(1966); 616-625.

Knowlton, Kenneth C. A programmer's description of L^6, Bell Telephone Laboratories' Low-Level Linked List Language. Murray Hill, N.J.: Bell Telephone Laboratories, Feb. 1966.

Knuth, D.E. and McNeley, J.L. SOL—A symbolic language for general-purpose system simulation. IEEE Trans. EC-13(1964); 401-408.

Knuth, D.E. and Merner, J.N. ALGOL 60 confidential. Comm. ACM, 4(1961); 268-272.

Knuth, D.E. and Wegner, P. Outline of a proposed undergraduate course on information structures. SICSAM Bulletin #1. Dec. 1965; 8-11.

Korolev et al. Soviet cybernetics technology: VIII. Report on the algorithmic language ALGEC. Doc. no. RM-5136-PR. RAND Corp., 1966. AD 644 869.

Krasnow, H.S. and Merikallio, R.A. The past, present and future of general simulation languages. Management Science, 11(1964).

Kulsrud, H.E. A general purpose graphic language. Comm. ACM, 11(1968); 247-254.

Lackner, Michael R. Digital simulation and system theory. Doc. no. SP-1612. Santa Monica, Calif.: SDC, 1964. AD 610 697.

Landaeta, William. An introduction to data structuring process. Doc. no. SP-2136. Santa Monica, Calif.: SDC, 1965.

Landin, P.J. A correspondence between ALGOL 60 and Church's Lambda-notation. Comm. ACM, 8(1965); 89-101, 158-165.

Landin, P.J. A formal description of ALGOL 60. IN: Steel, T.B., Jr. ed. Formal language description languages for computer programming; 266-294.

Landin, P.J. A lambda-calculus approach. IN: Fox, L., ed., Advances in programming and non-numerical computation. New York, Pergamon, 1966; 97-141.

Landin, P.J. The mechanical evaluation of expressions. Computer Journal, Jan. 1964; 308-320.

Lang, C.A., Polansky, R.B., and Ross, D.T. Some experiments with an algorithmic graphical language. Doc. no. MIT-ESL-TM-220. Cambridge, Mass.: M.I.T. Electronic Systems Laboratory, 1965. AD 472 147.

Larson, Robert P. Data filtering applied to information storage and retrieval applications. Comm. ACM, 9(1966); 785-789, 793.

Laski, J. Sets and other types. ALGOL Bulletin 27, Feb. 1968.

Lass, S.E. PERT time calculations without topological ordering. Comm. ACM, 8(1965); 172-174.

Laszlo, Steven I. Toward a semiotic theory of programming languages. Paramus, N.J.: Decision Systems Inc. Nov. 1967.

Leavenworth, B.M. Syntax macros and extended translation. Comm. ACM, 9(1966); 790-793.

Ledley, R.S., Jacobson, J., and Belson, M. BUGSYS: a programming system for picture processing--not for debugging. Comm. ACM, 9(1966); 79-84.

Leroy, H. A proposal for macro-facilities in ALGOL. ALGOL Bulletin #22, 1966; 15-26.

Levien, Roger, et al. Relational data file: a tool for mechanized inference execution and data retrieval. Doc. no. RM-4793-PR. Santa Monica, Calif.: RAND Corp., 1965. AD 625 409.

Levien, R.E. Studies in the theory of computational algorithms: 1. Formalization, computability, representation, and analysis problems. Doc. no. RM-3007-PR. Santa Monica, Calif.: RAND Corp., 1962. AD 270 816.

Levin, M. et al. LISP II project. Doc. series no. TM-2260. Santa Monica, Calif.: SDC, 1965.

Libbey, M.A. General results and recommendations: Part 1 of report of Phase I of D-60 Information Element Standardization Program. Doc. no. W-7080. Bedford, Mass.: The MITRE Corp., 1964.

Libbey, M.A. and Higgens, D.C. Detailed problem discussions: Part 2 of report of Phase 1 of D-60 Information Element Standardization Program. Doc. no. W-7081. Bedford, Mass.: The MITRE Corp., 1964.

Linder, W. Mathematical object description language. M.I.T. Masters thesis, 1965.

Lindsay, Robert K., Pratt, Terrence W., and Shavor, Kenneth M. An experimental syntax-directed data structure language. Doc. no. P-3112. Santa Monica, Calif.: RAND Corp., 1965. AD 614 782.

Loev, G., ed. SLIP list processor. Philadelphia, Pa.: University of Pennsylvania Computer Center, 1966.

Lombardi, L.A. On the declarative control of the data flow by means of recursive functions. IN: Symbolic Languages in Data Processing. New York: Gordon and Breach, 1962.

Lombardi, L.A. A general business-oriented language based on decision expressions. Comm. ACM, 7(1964); 104-111.

Lombardi, L.A. Mathematical structure of non-arithmetic data processing procedures. Jour. ACM, (1962); 136-159.

Lombardi, L.A. Non-procedural data system languages. SUMMARY IN: Preprints of Summaries of Papers Presented at the 16th National Meeting of the ACM, Los Angeles, 1961; paper 11-1.

Lombardi, L.A. System handling of functional operators. Jour. ACM, 8(1961); 168-185.

Lombardi, L.A. On table-operating algorithms. IN: Information Processing, 1962. Amsterdam: North-Holland, 1963; 509-512.

Lombardi, L.A. Theory of files. Proc. EJCC, 1960; 137-141.

Loomis, R.G. Boundary Networks. Comm. ACM, 8(1965); 44-48.

Love, H.H., Jr. and Savitt, D.A. The association-storing processor. IN: Shapiro and Rogers, eds. Prospects for simulation and simulators of dynamic systems. Proc. Baltimore Symp., 1966. New York: Spartan Books, and London: Macmillan & Co., Ltd., 1967; 165-182.

Lucas, P. On the formalization of syntax and semantics of PL/I. Doc. no. TR 25.060. Vienna: IBM Laboratory, 1965.

Lucas, P. Requirements on a language for logical data processing. IN: Information Processing, 1962. Amsterdam: North-Holland, 1963; 556-560.

Lukaszewicz, L. EOL—a symbol manipulation language. Computer Journal, May 1967; 53-59.

McCarthy, J. A basis for a mathematical theory of computation. IN: Braffort, P., and Hirschberg, D., eds. Computer Programming and Formal Systems. Amsterdam: North-Holland, 1963; 33-70.

McCarthy, J. and Painter, J. Correctness of a compiler for arithmetic expressions. Tech. Rept. no. CS38. Stanford University.: April 1966.

McCarthy, J. Definition of new data types in ALGOL X. ALGOL Bulletin #18 (1964); 45-46.

McCarthy, John. Problems in the theory of computation. IN: Information Processing 1965. Washington, D.C.: Spartan Books, 1965; vol. 1; 219-222.

McCarthy, Jr. Situations, actions, and causal laws. Stanford Artificial Intelligence Project Memo no. 2. Palo Alto, Calif.: Stanford Univ. July 1968.

McClure, Robert M. A programming language for simulating digital systems. Jour. ACM, 12(1965); 14-22.

McKeeman, W.M. An approach to computer language design. Doc. no. CS48. Stanford, Calif.: Stanford University Computer Science Dept.,1966. AD 639 166.

Mann, William C., and Jensen, Paul A. A data structure for directed graphs in man-machine processing. Washington, D.C.: Computer Command and Control Co., 1966. Doc. no. 77-106-1. AD 636 251.

Mann, William C. Language facilities for a man, a machine, and relational data. Doc. no. 106-2. Philadelphia, Pa.: Computer Command and Control Co., 1967. AD 651 973.

Markowitz, H.M., Hausner, B., and Karr, H.W. SIMSCRIPT, Asimulation programming language. Doc. RM-3310-PR. Englewood Cliffs, N.J.: Prentice-Hall, 1963. AD 291 806.

Maurer, Ward Douglas. Computer experiments in finite algebra. Comm. ACM, 9(1966); 598-603.

Mealy, G.H. et al. The functional structure of OS/360. IBM Systems Journal, 5(1966); 2-51.

Merner, J.N. A splitting of formatting from input-output. ALGOL Bulletin #22, 1966; 9-10.

Merrill, Roy, Jr. Some properties of ternary threshold logic. IEEE Trans. EC-13(1964).

Miller, R.H. An example in "significant-digit" arithmetic. Comm. ACM, 7(1964); 21-23.

Mitchell, Robert, Cheatham, T.E., Jr. and Christensen, Carlos. A basis for core language design. Wakefield, Mass.: MCA, 1967. Doc. no. CA 6701-2711.

Mitchell, R.W., Christensen, C., Myszewski, M., Sampson, Caral. An informal PL/I roundtable, collection one. Doc. no. CA-6704-0511. Wakefield, Mass.: Massachusetts Computer Associates, 1967; 35 pp.

Montalbano, M. Tables, flow charts and program logic. IBM Systems Journal, 1(1962); 51-63.

Mooers, C. TRAC, a procedure-describing language for the reactive typewriter. Comm. ACM, 9(1966); 215-219.

Mooers, C.N. TRAC, a text-handling language. Proc. ACM, Natl. Conference, 1965; 229-246.

Morris, C.W. Signs, language, and behavior. New York: Braziller, 1955.

Morris, N.I. A data structure programming language. M.I.T. B.A. thesis, 1965.

Mullery, A.P. A procedure-oriented machine language. IEEE Trans. EC-13 (1964); 449-455.

Napper, R.B.E. Some proposals for SNAP, a language with formal macro facilities. Computer Journal, 10(1967); 231-243.

Narasimhan, R. Programming languages and computers: a unified metatheory. Tech. Report #9. Bombay: Tata Institute of Fundamental Research, 1966.

Naur, P. The place of programming in a world of problems, tools and people. Paper presented at the IFIP Congress, N.Y., 1965.

Naur, P. et al. Revised Report on the algorithmic language ALGOL 60. Comm. ACM, 6(1963); 1-18.

Naylor, T.H., Balintfy, J.L., Burdick, D.S. Simulation languages. Chapter 7 IN: Computer Simulation Techniques, New York: John Wiley & Sons, Inc. 1966.

Nelson, T.H. A file structure for the complex, the changing and the indeterminate. ACM Proc., 1965; 84-100.

Newell, A., et al. Information processing language-V manual, 2nd edition. Englewood Cliffs, N.J.: Prentice-Hall, 1964.

Newell, A. On the representations of problems. IN: Computer Science Research Review. Pittsburgh, Pa.: Carnegie Institute of Technology, 1966; 19-33.

Nickerson, R.C. An engineering application of logic-structure tables. Comm. ACM, 4(1961); 516-520.

NICOL-Nineteen Hundred Commercial Language. IN: I.C.T. Data Processing Journal, no. 28 (1966); 26-29.

Oettinger, A.G. Linguistic problems of man-computer interactions. IBM report no. 320-1941-0. IN: Proc. IBM Scient. Comp. Symp., Man-Machine Comm. New York: 1965; 33-40.

Olle, T.W. Use of INFOL for document retrieval applications. IN: Proc. 1966 Annual Mtg. American Documentation Institute. Adrienne Press, 1966; 321-327.

Oppenheim, D.K. The META5 language and system. Doc. no. TM-2396. Santa Monica, Calif.: SDC, 1965.

Ore, O. Graphs and their uses. New Math Library, #10; New York: Random House, 1963.

Ostrand, Thomas Joseph. Interim Technical Report: an expanding computer operating system. Doc. no. 67-16. Philadelphia, Pa.: Moore School of Electrical Engineering, University of Pennsylvania. 1966.

Overheu, D.L. An abstract machine for symbolic computation. Jour. ACM, 13(1966); 444-468.

Parente, R.J. and Krasnow, H.S. A language for modeling and simulating dynamic systems. CACM, 10(1967); 559-567.

Parente, R.J. A language for dynamic system description. ASDD Tech. Rpt. no. 17-180. Yorktown Heights, N.Y.: IBM Advance Systems Development Division, 1965.

Parker, Donn B. Solving design problems in graphical dialogue. Computer Group News, 1966; 1-12.

Parnas, David L. A language for describing the functions of synchronous systems. Comm. ACM, 9(1966); 72-76, 79.

Peirce, C.S. (J. Buchler, ed.) The Philosophical Writings of Peirce. New York, Dover, 1955.

Perlis, Alan J., Iturriaga, Renato, and Standish, Thomas A. A definition of formula ALGOL. Pittsburgh, Pa.: Carnegie Institute of Technology, 1966.

Perlis, A.J. and Iturriaga, R. An extension to ALGOL for manipulating formulae. Comm. ACM, 7(1964); 127-130.

Perlis, Alan J., Iturriaga, Renato, and Standish, Thomas. A preliminary sketch of formula ALGOL. Pittsburgh, Pa.: Carnegie Institute of Technology, 1965.

Perlis, A.J. The synthesis of algorithmic systems. Proc. ACM, 1966; 1-6.

Perstein, Millard H. The JOVIAL (J3) grammar and lexicon. Doc. no. TM-555/002/04B. Santa Monica, Calif.: SDC, 1966. AD 623 861.

Perstein, Millard H. Numbered-line syntactic description of JOVIAL (J3). Doc. no. SP-2311/000/00. Santa Monica, Calif.: SDC, 1966.

Pigniolo, J.C. The programmed system automaton: a modular concept of the logical structure of computer programs. NEL Rpt. 1274. San Diego, Calif.: U.S. Navy Electronics Laboratory, 1965. AD 463 187.

Pratt, Terrence, W. Syntax-directed translation for experimental programming languages. Doc. no. TNN-41. Austin, Texas: University of Texas, Computation Center, 1965. Generalized data structures, p. 44.

Proceedings, IBM Scientific Computing Symposium, Digital simulation of continuous systems. IBM report no. 320-1943-0. Held in Yorktown Heights, N.Y.: 1966, June 20-22.

Prywes, N.S. and Gray, H.J. The multi-list system for real-time storage and retrieval. IN: Information Processing 1962. Amsterdam; North-Holland, 1963; 273-278.

Quillian, M. Ross. Semantic memory. Doc. no. AFCRL-66-189. Bolt, Beranek and Newman, Inc., 1966. AD 641 671.

Quillian, R. Word concepts: a theory and simulation of some basic semantic capabilities. Doc. no. SP-2199. Santa Monica, Calif.: SDC, 1965.

Radin, G., and Rogoway, H.P. NPL: highlights of a new programming language. Comm. ACM, 8(1965); 9-17.

Rajchl, J. An attempt to construct a formulational language for business problems. IN: Information Processing Machines, Proc. of Symposium held in Prague on Sept. 7-9, 1964. Czech. Acad. of Sciences (Prague) and Iliffe Books, Ltd. London: 1965; 132-139.

Raphael, B. Applications of symbolic manipulation. SICSAM Bulletin no. 1, Dec. 1965; 12-22.

Raphael, B., Bobrow, D.G., Fein, L., and Young, J.W. A brief survey of computer languages for symbolic and algebraic manipulation. Menlo Park, Calif.: Stanford Research Institute, 1966.

Raphael, B. SIR: a computer program for semantic information retrieval. Doc. no. MAC-TR-2. Cambridge, Mass.: Project MAC, Massachusetts Institute of Technology, 1964. AD 608 499.

Raphael, Bertram. The structure of programming languages. Comm. ACM, 9(1966); 67-71.

Reinwald, Lewis T. An introduction to TAB40: a processor for table-written FORTRAN IV programs. McLean, Va.: Research Analysis Corp., 1966. AD 647 418.

Rescher, N. The logic of commands. New York: Dover Publications, 1966.

Reynolds, J.C. COGENT programming manual. Doc. no. ANL-7022. Argonne, Ill.: Argonne National Laboratory, 1965.

Reynolds, J.C. Cogent 1.2 Operations Manual. Doc. no. CS37. Stanford, Calif.: Standford University, Computer Science Department, 1966.

Reynolds, J. An introduction to the COGENT programming system. Proc. 20th National Conference, ACM., 1965; 422-436.

Riley, John A. The algebra of block diagrams. Doc. no. AFCRL-65-805. Parke Mathematical Laboratories, Inc. 1965; AD 625 081.

Roberts, L.G. Graphical communication and control languages. IN: 2nd Conference on Information System Sciences, 1964. AD 632 587.

Rohrbacker, Donald L. Advanced computer organization study, vol. 1. Basic report. Akron, Ohio: Goodyear Aerospace Corp., 1966. AD 631 870. Vol 2, Appendixes, AD 631 871.

Rome, S.C., and Rome, B.K. Formal representation of intentionally structured systems. IN: Kent, A., ed. Information Retrieval and Machine Translation, part 1. New York: Interscience, 1960: 467-492.

Rosenthal, L.E. Analytic algebraic manipulation. Computer Journal, 19 (1967); 265-270.

Rosin, Robert F. Macros in PL/I. PL/1 Bulletin No. 2, Aug. 1966; 16-22.

Ross, Douglas T. The AED approach to generalized computer-aided design. Doc. no. ESL-R-305. Massachusetts Institute of Technology, 1967. AD 814 812.

Ross, Douglas T. AED, Jr.: an experimental language processor. Doc. no. ESL-TM-211. Cambridge, Mass.: M.I.T. Electronics Systems Laboratory, 1964. AD 453 881.

Ross, Douglas T. An algorithmic theory of language. Doc. no. ESL-TM-156. Cambridge, Mass.: M.I.T. Electronics Systems Laboratory, 1962. AD 296 998.

Ross, D.T. A generalized technique for symbol manipulation and numerical calculation. Comm. ACM, 4(1961); 147-150.

Ross, D.T. Implications of computer-aided design for numerically controlled production. Doc. no. ESL-TM-212. Cambridge, Mass.: M.I.T. Electronics Systems Laboratory, 1964. AD 453 880.

Ross, D.T. and Rodrigues, J.E. Theoretical foundations for the computer-aided design system. Proc. SJCC, 1963; 305-322.

Rovner, P.D., and Feldman, J.A. An associative processing system for conventional digital computers. Doc. no. 1967-19. ESD-TR-67-242. Lexington, Mass.: M.I.T. Lincoln Laboratory, 1967. AD 655 810.

Rovner, P.D. and Feldman, J.A. The LEAP language and data structure. Proc. IFIP Congress, Edinburgh, 1968.

Rutishauser, H. Contributions to the discussion of ALGOL 6x. ALGOL Bulletin #19(1965); 44-50.

Ruyle, A., Brackett, John W., and Kaplow, Roy. The status of systems for on-line mathematical assistance. Proc., ACM, 1967; 151-167.

Sable, J., et al. Design and reliability central data management system. RADC-TR-65-189, vol. 1. Rome, N.Y.: RADC, c. 1966. AD 620 025.

Sable, J., et al. Design of reliability central data management system. Doc. no. RADC-TR-65-189, vol. 2. Grifiss AFB, New York: RADC, 1965. AD 469 269.

Sable, Jerome D. Language and information structure in information systems. Ph.D. dissertation, 1963. Philadelphia, Pa.: Moore School of Electrical Engineering, University of Pennsylvania.

Sable, J., et al. Reliability central automatic data processing system. Doc. no. RADC-TR-66-474. Griffiss, AFB, New York: RADC, 1966. AD 489 668.

Sakoda, J.M. DYSTAL manual. Providence, R.I.: Brown University, Sociology Computer Laboratory, 1965.

Salton, G. Manipulation of trees in information retrieval. Comm. ACM, 5(1962); 103-114.

Samelson, K. Functionals and functional transformation. ALGOL Bulletin #20 (1965); 27-28.

Sammet, J.E. An annotated descriptor based bibliography on the use of computers for non-numerical mathematics. Computing Reviews, July-Aug., 1966; B-1 - B-31.

Sammet, Jean E. Formula manipulation compiler. Datamation, July 1966; 32-34, 39-41.

Sammet, J.E. Survey of formula manipulation. Comm. ACM, 9(1966); 555-569.

Sammet, J.E. Survey on the use of computers for doing non-numerical mathematics. Doc. no. IBM TR00.1428. Poughkeepsie, N.Y.: IBM Systems Division, 1966.

Sansom, F.J. and Peterson, H.E. MIMIC programming manual. Doc. no. SEG-TR-67-31. Wright-Patterson AFB, Ohio: Aeronautical Systems Division, Air Force Systems Command, 1967. AD 656 301.

Satterthwait, Arnold C. Programming languages for computational linguistics. IN: Alt, F., ed. Advances in Computers, vol. 7. New York: Academic Press, 1966; 209-238.

Savitt, D.A. et al. Association-storing processor study. Doc. no. RADC-TR-66-174. Griffiss AFB, N.Y.: RADC, 1966. AD 488 538.

Schlaeppi, H.P. A formal language for describing machine, logic, timing, and sequencing (LOTIS). IEEE Trans. EC-13(1964); 439-448.

Schmidt, D.T. and Kavanagh, T.F. Using decision structure tables: part 1, principles and preparation. Datamation, 1964; 42-52.

Schmidt, D.T., and Kavanagh, T.F. Using decision structure tables: part 2, manufacturing applications. Datamation, 1964; 52-54.

Schwartz, J.I. Comparing programming languages. Computers and Automation, Feb. 1965; 15-16, 26.

Schwartz, J.I. On-line programming languages. Doc. no. SP-2048. Santa Monica, Calif.: SDC, 1965.

Seegmuller, G. Some proposals for ALGOL X. ALGOL Bulletin #21 (1965); 7-22. Corrections, ALGOL Bulletin #22 (1966); 5.

Seegmuller, G. Toward a grasp of procedure in algorithmic languages. Munich: Technische Hochschule, 1966. (In German).

Shaw, C.J. A bibliography for generalized information system designers. Doc. no. TM-2289. Santa Monica, Calif.: SDC, 1965.

Shaw, Christopher J. Computer software: developments and recent trends in programming and operating trends. SP-2582/000/00. SDC, 1966.

Shaw, C.J. On declaring arbitrarily coded alphabets. Comm. ACM, 7(1964); 288-290.

Shaw, C.J. Further U.S.A. work affecting commercial programming languages. Presented at the Congress of the International Federation for Information Processing, New York City: May 24-29. Doc. no. SP-2030. Santa Monica, Calif.: SDC, 1965.

Shaw, Christopher, J. Metalinguistic and numeric parameters and functions in language description. SIGPLAN Notices, March, 1968; 19-24.

Shaw, C.J. A specification of JOVIAL. Comm. ACM, 6(1963); 721-736.

Shaw, C.J. Theory, Practice, and trend in business programming. Doc. no. SP-2030/000/01. Santa Monica, Calif.: SDC, 1965.

Shaw, C. The utility of PL/I for command and control programming. Doc. no. SP-3014. Santa Monica, Calif.: SDC, 1968.

Shooman, W. Parallel computing with vertical data. Proc. EJCC, 1960; 111-115.

Sibley, R.A. The SLANG system. Comm. ACM, Jan., 1961.

Simmons, R.F. Answering English questions by computer: a survey. C ACM, 8(1965); 53-70.

Simmons, R.F. Storage and retrieval of aspects of meaning in directed graph structures. Doc. no. SP-1975/001/02. Santa Monica, Calif.: SDC, 1965.

Simon, Herbert, A. Experiments with a heuristic compiler. Jour. ACM, 10(1963); 493-506.

Singer, Theodore. Summary report of a workshop on programming languages for command and control. IEEE Trans. on Electronic Comp., 1966; 131-132.

Slotnick, Daniel L., Borck, W. Carl, and McReynolds, Robert C. The SOLOMON computer. Proc. FJCC, 1962; 97-107.

Smith, Donald L. Models and data structures for digital logic simulation. Doc. no. MAC-TR-31. Cambridge Mass.: M.I.T. Project MAC, 1966. AD 637 192.

Smith, L.W. Information and transformations on general arrays. SUMMARY IN: Preprints on Summaries of Papers Presented at the 16th National Meeting of ACM, Los Angeles, 1961; Paper 6B-3.

Smith, Peter J. Symbolic derivatives without list processing, subroutines, or recursion. Comm. ACM, 8(1965); 494-500.

Spector, C. Theory of programming. Doc. no. TM-570. Santa Monica, Calif.: SDC, 1960.

Squires, B.E. Compiler systems and metalanguages. Report no. 208. University of Illinois, Aug. 1966.

Srinavasan, C.V. Formal definition of CDL1, a computer description language. AFCRL-67-0588. Air Force Cambridge Research Labs. AD 662 899.

Standish, Tim (Thomas A.) On formula ALGOL and the evolution of programming languages. IN: Computer Science Research Review, 1967. Pittsburgh, Pa.: Carnegie-Mellon University; 9-17.

Standish, Thomas A. A data definition facility for programming languages. Doctoral dissertation. Pittsburgh, Pa.: Carnegie Institute of Technology, Dept. of Computer Science, May 1967; 292 pp. AD 658 042.

Steel, T.B., Jr. Beginnings of a theory of information handling. Comm. ACM, 7(1964); 97-103.

Steel, T.B. Jr, ed. Formal language description languages for computer programming. Amsterdam: North-Holland Publ. Co., 1966.

Steel, T.B., Jr. A formalization of semantics for programming language description. Doc. no. TM-2043/000/00. Santa Monica, Calif.: SDC, 1965.

Steel, T.B. Jr. The foundations of a theory of information processing. Doc. no. FN-6341. Santa Monica, Calif.: SDC, 1962.

Steel, T.B. Jr. Procedure languages for military command and control. IN: Proceedings of the Symposium onComputer Programming for Military Systems, The Hague, 28 Sept.- 2 Oct., 1964. The Hague, Netherlands: SHAPE Technical Centre, 1965; vol. 1.; 261-269.

Steel, T.B., Jr. Some observations on the relationship between JOVIAL and PL/I. Doc. no. TM-2930/000/01. Santa Monica, Calif.: SDC, 1966.

Stockham, T.G. Jr. Some methods of graphical debugging. IBM rep. no. 320-1941-0. IN: IBM Scient. Comp. Symp., Man-Machine Communication. New York: 1965; 57-71.

Stone, A.J. Phoenix compiler language and software system. IN: Proc. of the 1st Spaceborne Computer Software Worl shop. 1966; 223-233.

Strachey, C. A general purpose macrogenerator. Comp. Jour., 8(1965); 225-241.

Strachey, C., and Wilkes, M.V. Some proposals for improving the efficiency of ALGOL 60. Comm. ACM, 4(1961); 488.

Strachey, C. Towards a formal semantics. IFIP Working Conf. on Formal Language Description Languages, Baden: 1964.

Strauss, J.C. and Gilbert, W.L. SCADS: a programming system for the simulation of combined analog digital systems. Pittsburgh, Pa.: Carnegie Institute of Technology, 1964. (3rd ed.)

Sussenguth, E.H. Structure matching in information processing. Ph.D. thesis, Harvard University, 1964.

Sutherland, I. Sketchpad, A man-machine graphical communication system. Doc. no. TR-296. Lexington, Mass.: M.I.T. Lincoln Laboratory, 1963.

Sutherland, I.E. Sketchpad: a man-machine communication system. Proc. SJCC, 1963.

Sutherland, W.R. On-line graphical specification of computer procedures. Doc. no. ESD-TR-66-211. Lincoln Laboratory, 1966. AD 639 734.

Sutherland, W.R. The CORAL language and data structure. Exerpt (App.C) from "The On-Line Graphical Specification of Computer Procedures" M.I.T. Doctoral thesis, to be printed as Lincoln Lab. Technical Report 405.

Syn, W.M. and Wyman, D.G. DSL/90: Digital simulation language user's guide. Doc. no. San Jose TR 02.355. San Jose, Calif.: IBM Corp., 1965.

Syn, W.M., and Linebarger, Robert N. DSL/90—a digital simulation program for continuous system modeling. Proc. SJCC, 1966; 165-187.

Systems Development Corp. Report on initial planning for GENISYS (Generalized Information System). Doc. no. TM(L)-2335/000/01. ESD-TR-65-463. Santa Monica, Calif.: SDC, 1965. AD 472 209.

System Development Corp. Baum, C. ed. Second symposium on computer-centered data base systems. Doc. no. TM-2624/100/00. Santa Monica, Calif.: SDC, 1965. AD 625 417.

System Development Corp., Santa Monica. Spaceborne software systems study: vol. 3, recommendation for a common space programming language. Doc. no. SSD-TR-67-11, vol. 3 Los Angeles: AF Systems Command, Space Systems Division, Jan. 1967. AD 807 399L.

Systems Research Group. MILITRAN programming manual. Doc. no. ESD-TR-64-320. Mineola, N.Y.: Systems Research Group, 1964. AD 601 796.

Systems Research Group. MILITRAN reference manual. Doc. no. EST-TDR-64-390. Mineola, N.Y.: Systems Research Group, 1964. AD 601 794.

Tabory, R. Specifications for a tree processor. Yorktown Heights, N.Y.: IBM Research Center, c. 1965.

Tabory, R. Survey and evaluation of AED system at M.I.T. Report no. TR 00.1383. IBM, 1966.

Teichroew, Daniel, and Lubin, John Francis. Computer simulation—discussion of the technique and comparison of languages. Comm. ACM, 9(1966); 723-741.

Teitelman, W. FLIP—a format list processor. Doc. no. MAC-M-263. Cambridge, Mass.: M.I.T. Project MAC, 1966.

Teitelman, W. PILOT, a step toward man-computer symbiosis. Doc. no. MAC-TR-32. Cambridge, Mass.: M.I.T. Project MAC, 1966. AD 638 446.

Thompson, Frederick B. English for the computer. Proc. FJCC, 1966; 349-356.

Tobey, R.G. Experience with FORMAC algorithm design. Comm. ACM, 9(1966); 589-597.

Tocher, K.D. Review of simulation languages. Operational Research Quarterly, 16(1965); June.

Toda, M, and Shuford, E.H., Jr. Logic of systems: introduction to the formal theory of structure. Doc. no. ESD-TDR-64-193. Bedford, Mass.: Decision Sciences Laboratory, Air Force Electronic Systems Division, AD 432 879.

Unger, Stephen H. GIT--a heuristic program for testing pairs of directed line graphs for isomorphism. Comm. ACM, 7(1964); 26-34.

van der Riet, R.P. Formula manipulation in ALGOL 60. Amsterdam: Mathematisch Centrum, 1966. Doc. no. TW 101. AD 801 443.

Van Wijngaarden, A., ed. Draft report on the algorithmic language ALGOL 68. Report no. MR93. Amsterdam: Mathematisch Centrum, 1968.

Van Wijngaarden, A. Generalized ALGOL. IN: Annual Review in Automatic Programming, R. Goodman, ed. vol. 3, 1963; 17-26.

Walker, D.E., and Bartlett, J.M. The structure of languages for man and computer: problems in formalization. Paper presented at the First Congress on the Information System Sciences, 1962.

Wegner, P., ed. Introduction to systems programming. New York: Academic Press, 1964.

Wegner, Peter. The relation between the lambda calculus and programming languages. IN: Wegner, P. Some Theoretical Concepts in Programming, a tutorial paper.

Wegner, P. Some theoretical concepts in programming: a tutorial paper. University Park, Pa.: Computer Science Dept., Pennsylvania State University, Nov. 1965.

Wegner, P. What is special about languages for algebraic manipulation? SICSAM Bulletin no. 2, Feb, 1966; 7-13.

Weinberg, G.M. PL/I programming primer. New York: McGraw-Hill, 1966.

Weissman, C. LISP primer: a self-tutor for Q-32 LISP 1.5 Doc. no. TM-2337/010/00. Santa Monica, Calif.: SDC, 1965.

Weizenbaum, Joseph. Symmetric list processor (SLIP) Comm. ACM, 6(1963).

Wells, M.B. Aspects of language design for combinatorial computing. IEEE Trans. EC-13(1964); 431-438.

Wells, M.B. MADCAP: a scientific compiler for a displayed formula textbook language. Comm. ACM, 4(1961); 31-36.

Wessler, Barry D. TracD, a graphic programming language. Masters thesis, Cambridge, Mass.: M.I.T. May 1967.

White, G.W.T. Digital simulation languages for the solution of process control problems. IN: Proc., IBM Scientific Computing Symposium, Digital Simulation of Continuous Systems; 71-81.

Whitney, G.E. An analysis of the syntax of the programming language PL/I. Hopewell, N.J.: Western Electric Engineering Research Center, 1966.

Wigler, K. Evaluation of MILITRAN programming language. Naval Command Systems Support Activity, May 1966.

Wilkes, M.V. Constraint-type statements in programming languages. Comm. ACM, 7(1964); 587-589.

Williams, Leland, H. Algebra of polynomials in several variables for a digital computer. Jour. ACM, 9(1962); 29-40.

Williams, R.G., and Wishner, R.F. Compiler language and standardization and evaluation. Doc. no. RAC-TP-138. McLean, Va.: Research Analysis Corp., 1964. AD 458 448.

Williams, T.M., Barnes, R.F., and Kuipers, J.W. Discussion of major features of a restricted logistic grammar for topic representation. Doc. no. II-5206-26. Lexington, Mass.: Itek Laboratories, Feb. 1962.

Wirth, N., and Weber, H. EULER: A generalization of ALGOL and its formal definition. Comm. ACM 9(1966); 13-25, 89-99. Originally issued in Tech. Rpt. CS20 by Computer Science Dept.: Stanford University, 1965.

Wirth, N. On certain basic concepts of programming languages. Doc. no. CS65. Stanford, Calif.: Stanford University Dept. of Computer Science, May 1965. 33pp.

Wirth, N. A proposal on string manipulation in ALGOL 60. ALGOL Bulletin #17 (1964); 13-17.

Wolman, B.L. Operators for manipulating language structures. Doc. no. 9442-M-160. Cambridge, Mass.: M.I.T. Electronics Systems Lab., 1966.

Woodger, M. ALGOL X, Note on the proposed successor to ALGOL 60. ALGOL Bulletin #22 (1966); 28-33.

Yershov, A.P. ALPHA—an automatic programming system of high efficiency. Jour. ACM, 13(1966); 17-24.

Yngve, V. COMIT Programmer's reference manual. Cambridge, Mass.: M.I.T., 1961.

Young, J.W., Jr. and Kent, H.K. Abstract formulation of data processing problems. Journal of Industrial Engineering, Nov/Dec 1958; 471-479.

Young, J.W., Jr. Non-procedural languages—a tutorial. Paper presented at ACM So. Calif. Chapters 7th Annual Tech. Symp., March 1965.

Zarimba, W.A. On ALGOL I/O conventions. Comm. ACM, 8(1965); 167–169.

Ziehe, T.W. An organizational form for item management. Doc. no. TRACOR 67–1111–U. Austin, Texas: TRACOR, Inc., Feb. 1968. AD 665 981.

Zilli, Marisa Venturini. λ-K-formulae for vector operators. ICC Bulletin 4, 3(July–Sept. 1965), 157–174.

Zwicky, A.M., Jr. and Isard, S. Some aspects of tree theory. Doc. no. W–6674. Bedford, Mass.: The MITRE Corp., 1963.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

University City Science Center
3401 Market Street
Philadelphia, Pa. 19104

2a. REPORT SECURITY CLASSIFICATION

unclassified

2b. GROUP

3. REPORT TITLE

EXTENSION OF PROGRAMMING LANGUAGE CONCEPTS

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

scientific; ; final

5. AUTHOR(S) *(First name, middle initial, last name)*

Philip R. Bagley

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
November 1968	217	485

8a. CONTRACT OR GRANT NO. F44620-67-C-0021	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO. 9769-05	
c. 6144501F	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* AFOSR 69-0023TR
d. 681304	

10. DISTRIBUTION STATEMENT

This document is approved for public sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Air Force Office of Scientific Research Directorate of Information Sciences Arlington, Va. 22209

13. ABSTRACT

This study concerns the extension of concepts used in current computer programming languages. The aim is to find ways of designing new programming languages which have increased flexibility without also having increased complexity. The key to accomplishing this is by generalizing on the current concepts. The work is based on the idea that it is possible to design a language which is truly independent of the hardware characteristics of current computers. In the course of the study, considerable re-examination of current concepts such as variables, procedure call mechanisms, and program sequence controls, has been required. A new technique of expressing data values, data elements, and data structures has been developed. The technique provides for the construction of arbitrarily-complex data elements, and for arbitrarily-chosen relationships between data elements. All expressions in a program which cause the language processor to take some action, which includes "declarations", are viewed as transformations ("procedures"). A basic set of these transformations has been proposed. The most significant demand on machine design which arises from this research is that much more freedom of storage organization is needed than is provided by conventional machines. Large-scale associative memories could be used to provide some of this needed flexibility of storage. Recommendations for further work are presented and an extensive bibliography on programming language, concepts, and design is appended.

DD FORM 1 NOV 65 1473

UNCLASSIFIED

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
programming language						
generalization of language concepts						
data elements						
data structures						
transformations						
language processors						
interpreters						
hardware-independence						
bibliography						

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
University City Science Center 3401 Market Street Philadelphia, Pa. 19104	unclassified
	2b. GROUP

3. REPORT TITLE

EXTENSION OF PROGRAMMING LANGUAGE CONCEPTS

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

scientific; ; final

5. AUTHOR(S) (First name, middle initial, last name)

Philip R. Bagley

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
November 1968	217	485

8a. CONTRACT OR GRANT NO. F44620-67-C-0021	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO. 9769-05	
c. 6144501F	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d. 681304	AFOSR 69-0023TR

10. DISTRIBUTION STATEMENT

This document is approved for public sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
TECH OTHER	Air Force Office of Scientific Research Directorate of Information Sciences Arlington, Va. 22209

13. ABSTRACT

This study concerns the extension of concepts used in current computer programming languages. The aim is to find ways of designing new programming languages which have increased flexibility without also having increased complexity. The key to accomplishing this is by generalizing on the current concepts. The work is based on the idea that it is possible to design a language which is truly independent of the hardware characteristics of current computers. In the course of the study, considerable re-examination of current concepts such as variables, procedure call mechanisms, and program sequence controls, has been required. A new technique of expressing data values, data elements, and data structures has been developed. The technique provides for the construction of arbitrarily-complex data elements, and for arbitrarily-chosen relationships between data elements. All expressions in a program which cause the language processor to take some action, which includes "declarations", are viewed as transformations ("procedures"). A basic set of these transformations has been proposed. The most significant demand on machine design which arises from this research is that much more freedom of storage organization is needed than is provided by conventional machines. Large-scale associative memories could be used to provide some of this needed flexibility of storage. Recommendations for further work are presented and an extensive bibliography on programming language, concepts, and design is appended.

DD FORM 1 NOV 65 1473

UNCLASSIFIED

Security Classification